

YEROTH_QVGE	2
YEROTH-ERP-3-0-SOFTWARE-SYSTEM-ARCHITECTURE_no_q- uality_assurance	14
YEROTH-ERP_multi_sites_base_de_donnees	52
yr-sd-db-runtime-verif	71

YEROTH_QVGE-user-guide	2
YEROTH_QVGE-intro	9

User's Guide for the Design and Testing System YEROTH_QVGE (YR_QVGE)



Figure 1: Portrait of PROF. DR.–ING. DIPL.–INF. XAVIER NOUNDOU .
Contact: yeroth.d@gmail.com

Contents

1	Introduction	3
2	YEROTH_QVGE (YR_QVGE) Short Overview	3
3	YEROTH_QVGE (YR_QVGE) Project Dependency	3
4	Advantages of YEROTH_QVGE	4
5	State Diagram Mealy Machine (SDMM)	4
5.1	HOW TO READ A "SDMM"	4
5.2	"SDMM" WITH MORE THAN 2 STATES	4
6	YEROTH_QVGE (YR_QVGE) Workflow	4
7	Custom User Project (YR-DB-RUNTIME-VERIF)	5
8	HOW TO START YR-DB-RUNTIME-VERIF	5
9	SQL QUERY Recovery execution on demand	6
9.1	Automatic SQL Command Query Generation	6
9.1.1	ERROR ACCEPTING STATE for sdm 1.	6
9.1.2	RECOVERY 1.	6
9.1.3	RECOVERY 2 (Practical solution to be implemented in YR-DB-RUNTIME-VERIF.	6
9.1.4	Concrete RECOVERY 2 action.	6
10	Formal Scientific and Engineering Project Description	6
11	Conclusion	6

Table 1: STATE DIAGRAM MEALY MACHINE SPECIFICATION KEYWORDS in YEROTH_QVGE

scientific keywords	engineering keywords
in_set_trace	in_sql_event_log
not_in_set_trace	not_in_sql_event_log
recovery_sql_query	recovery_sql_query
STATE	STATE
START_STATE	BEGIN_STATE
FINAL_STATE	END_STATE / ERROR_STATE
IN_PRE	IN_BEFORE
IN_POST	IN_AFTER
NOT_IN_PRE	NOT_IN_BEFORE
NOT_IN_POST	NOT_IN_AFTER

Figure 2: A motivating example, as previous bug found in YEROTH-ERP-3.0.

$Q_0 := \text{NOT_IN_BEFORE}(\text{YR_ASSET}, \text{department.department_name}).$

$Q_1 := \text{IN_AFTER}(\text{YR_ASSET}, \text{stocks.department_name}).$



Figure 3: A SAMPLE state diagram mealy machine file.

```

1. yr_sd_mealy_automaton_spec yr_missing_department_NO_DELETE
2. {
3.   START_STATE(d) : NOT_IN_BEFORE(YR_ASSET, department.department_name)
4.   -> [in_sql_event_log('DELETE.department.YR_ASSET', STATE(d))] / 'SELECT.department' ->
5.     ERROR_STATE(e) : IN_AFTER(YR_ASSET, stocks.department_name) .
6. }
  
```

Figure 4: A SCREENSHOT OF YEROTH_QVGE.

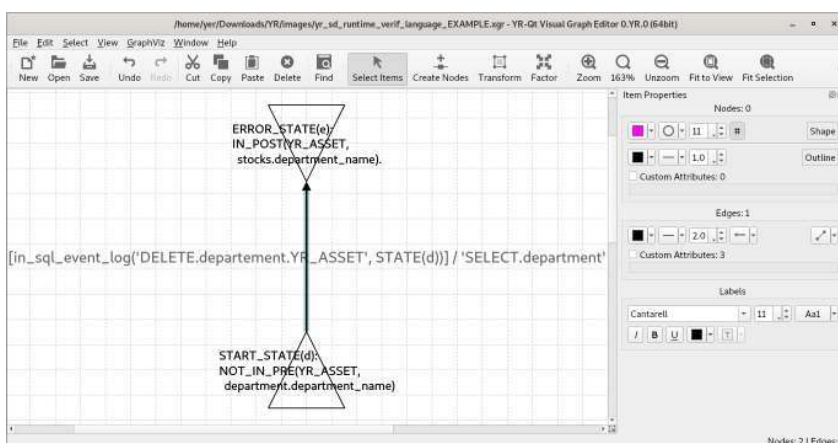


Figure 5: A SCREENSHOT OF YR-DB-RUNTIME-VERIF SQL EVENT LOG.

Time Stamp	SQL event log	source	target	changed qty
06:15:30.005	SELECT asset_id FROM YR_ASSET	YR-DB-RUNTIME-VERIF	YR-DB-RUNTIME-VERIF	1
06:15:30.004	SELECT asset_id FROM YR-DB-RUNTIME-VERIF	YR-DB-RUNTIME-VERIF	YR-DB-RUNTIME-VERIF	1

TIME STAMP	SQL recovered executed query
06:15:30.005	SELECT asset_id FROM YR_ASSET

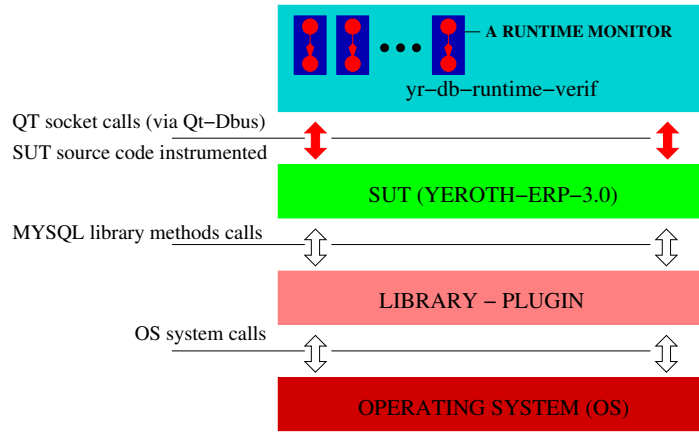
source file	line number
src/windows/stock/yeroth-erp-stocks-window-app	2149

before pre-condition on source state	after post-condition on target state
not_in_before(YR_ASSET, department.department_name)	in_after(YR_ASSET, stocks.department_name)

evaluated guarded condition expression / value	previous state	accepting state	is error state
in_sql_event_log('DELETE.department.YR_ASSET', STATE(d))	1	0	Yes

1 Introduction

Figure 6: SOFTWARE ARCHITECTURE OF YR-DB-RUNTIME-VERIF.



This user's guide helps briefly and concisely how to create a binary executable of the runtime monitoring testing tool **YR-DB-RUNTIME-VERIF** having user defined runtime monitors. The guide also specifies keywords allowed within runtime monitor specifications as State Diagram Mealy Machines.

YEROTH_QVGE (YR_QVGE) could be used for the following automatic generation, analysis, verification, and validation tasks:

1. Automatic generation of runtime monitoring module program to prove whether a test procedure, automated, or not, is correct with regards to a test and / or design STATE DIAGRAM MEALY MACHINE.

In effect, let the test execution be runtime monitored to watch whether accepting error states would be found.

For instance, Junit testing environment could automatically integrate an automatically generated runtime monitor infrastructure for unit testing.

2. Automatic generation of runtime monitoring module program for any software that can emit DBus messages.

Such runtime monitoring modules are for interest for special LTL model checking properties that cannot get a definite answer through use of a conventional model checker.

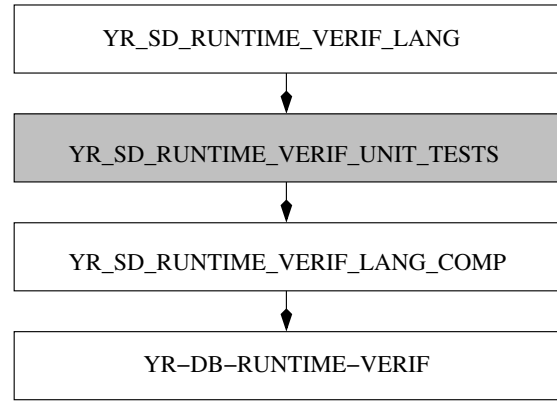
3. Software design properties with SQL
4. Software design properties including event sequences over different layers of software system architecture
5. Class diagram with sequence diagram.

2 YEROTH_QVGE (YR_QVGE) Short Overview

Figure 7: YEROTH_QVGE software library dependencies.

¹https://github.com/yerothd/yr_sd_runtime_verif

²<https://github.com/yerothd/yr-db-runtime-verif>



YEROTH_QVGE is a CASE (Computer-Aided Software Engineering) design tool to generate "domain-specific language (DSL) **YR_SD_RUNTIME_VERIF_LANG** ¹" files, to be inputted into the "compiler **YR_SD_RUNTIME_VERIF_LANG_COMP**", so to generate C++ files for the "runtime verifier tester **YR-DB-RUNTIME-VERIF** ²" that allows for manual verification of SQL correctness properties of Graphical User Interface (GUI) software.

Figure 8 illustrates a workflow diagrammatically of the afore described process.

Figure 7 show a diagram of the afore described process; The step of the unit tests is colored in gray because it is only for developers of YEROTH_QVGE intended.

YR-DB-RUNTIME-VERIF inputs SQL correctness properties expressed using the formalism "*state diagram mealy machine* (**YR_SD_RUNTIME_VERIF_LANG**)". Figure 6 illustrates a software system architecture of **YR-DB-RUNTIME-VERIF**, together with the monitored program under analysis. The Free Open Source Code Software (FOSS) tool-chain of development testing is located as follows for free, EXCEPT for "YEROTH_QVGE" that is a Closed Source Code Software (CSCS):

- **COMPILER** (i.e.: **YR_SD_RUNTIME_VERIF_LANG_COMP**): https://github.com/yerothd/yr_sd_runtime_verif_lang
- **RUNTIME VERIFIER TESTER** (i.e.: **YR-DB-RUNTIME-VERIF**): <https://github.com/yerothd/yr-db-runtime-verif>
- **state diagram mealy machine UNIT TESTS CODE** (i.e.: **YR_SD_RUNTIME_VERIF_UNIT_TESTS**): https://github.com/yerothd/yr_sd_runtime_verif_UNIT_TESTS
- **state diagram mealy machine** (i.e.: **YR_SD_RUNTIME_VERIF_LANG**): https://github.com/yerothd/yr_sd_runtime_verif

3 YEROTH_QVGE (YR_QVGE) Project Dependency

Table 2: YEROTH_QVGE Design and Testing System Dependencies

PROJECT	Required Library
1) YR_SD_RUNTIME_VERIF_LANG	
2) YR_SD_RUNTIME_VERIF_LANG_COMP	1)
3) YR_SD_RUNTIME_VERIF_UNIT_TESTS	1)
4) YR-DB-RUNTIME-VERIF	2)

Table 2 illustrates for each library project, which others it depends on.

4 Advantages of YEROTH_QVGE

A sample state diagram mealy machine is shown in Figure 3.

WITH manual drawing of SQL CORRECTNESS PROPERTY MODEL, you are freed from manually writing "state diagram mealy machine text files" that could be tedious and lengthy. Also, editing state diagram mealy machine files manually could be more error-prone than letting a compiler (YR_SD_RUNTIME_VERIF_LANG) do it for you.

5 State Diagram Mealy Machine (SDMM)

TABLE 1 depicts scientific keywords and their engineering counterpart that can be used in describing NOT DESIRABLE³ SQL⁴ call sequence state diagram mealy machine in YEROTH_QVGE Design and Testing System.

A STATE DIAGRAM mealy machine specification is compiled into C++ code that describes a runtime monitor to be executed in the runtime monitoring tester YR-DB-RUNTIME-VERIF. Figure 3 depicts a sample State Diagram Mealy Machine specification on a NOT DESIRABLE SQL call sequence.

5.1 HOW TO READ A "SDMM"

Figure 2 shows a finite automaton representation of the mealy machine description in Figure 3. It shall be read as follows:

- The program is in a start state D ; state D is a start state since there is incoming "START" arrow into it.
- (Pre-) Condition Q_0 : "department name 'YR_ASSET' is not in table column 'department_name' of database table 'department'"; applies in state D .
- Whenever GUARD CONDITION : ***in_sql_event_log('DELETE.department.YR_ASSET', STATE(d))***: "event 'DELETE.department.YR_ASSET' appears in SQL event log (trace) leading to state D "; applies in state D , system under test (SUT) event 'SELECT.department' could occur.
- When SUT event 'SELECT.department' occurs, SUT is now in state E ; state E is an error state because the node that represents it in Figure 2 has 2 circles on it.
- (Post-) Condition $\overline{Q_1}$: "department name 'YR_ASSET' is in table column

'department_name' of database table 'stocks'"; applies in state E .

This shall not be the case since department 'YR_ASSET' is no more defined in SUT database table 'department'.

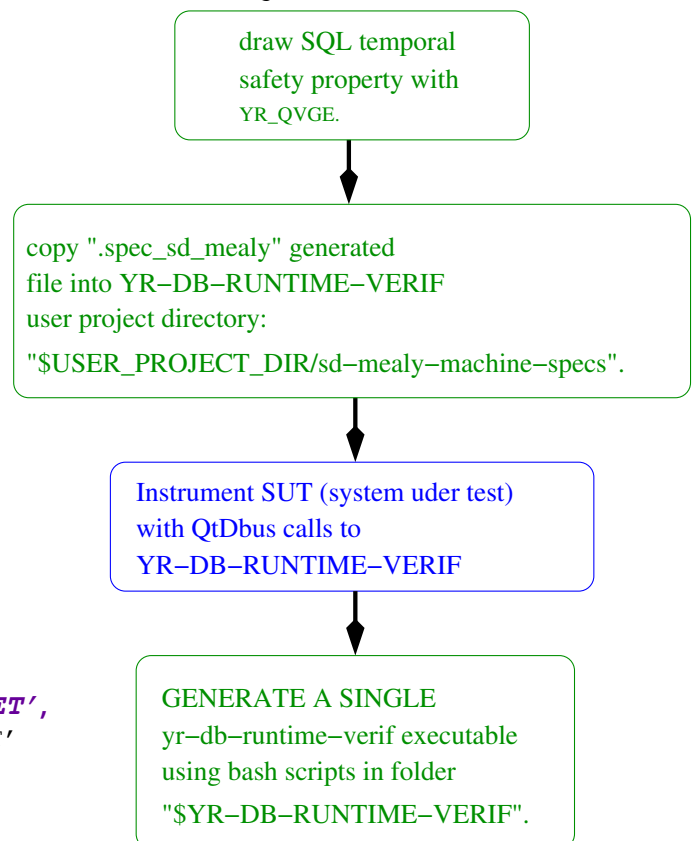
5.2 "SDMM" WITH MORE THAN 2 STATES

State Diagram Mealy Machines (SDMM) with more than 2 states have following characteristics, as detailed in scientific and engineering journal paper [Nou23] in preparation:

- Only the first transition has a pre-condition specification
- Each other transition only has a post-condition specification
- Since each state only has 1 outgoing state transition, the post-condition of the previous (incoming) state transition acts as the pre-condition of the next transition.

6 YEROTH_QVGE (YR_QVGE) Workflow

Figure 8: Workflow.



The "Design and Testing System" YEROTH_QVGE works with following workflow, as illustrated graphically in Figure 8:

1. Draw Structure Query Language (SQL) temporal safety property using drawing tool YEROTH_QVGE;

³Scientific: fail (forbidden) trace.

⁴Structure Query Language.

- copy the generated ".spec_sd_mealy" files into a user project directory in YR-DB-RUNTIME-VERIF home development folder: "\$YR-DB-RUNTIME-VERIF";
- follow the steps described in Section 7 so to gather a single executable that defines all specified runtime monitors.

7 Custom User Project (YR-DB-RUNTIME-VERIF)

Table 3: YR-DB-RUNTIME-VERIF Directories

Variable for illustration purposes	Meaning
\$YR-DB-RUNTIME-VERIF	root directory of YR-DB-RUNTIME-VERIF
\$YR-DB-RUNTIME-VERIF/\$USER_PROJECT	root directory of user project

Table 3 illustrates directories that will be used to describe a process to generate a single binary executable for a user's custom project with several runtime monitor specifications.

Figure 5 illustrates a screenshot of the Graphical User Interface (GUI) of YR-DB-RUNTIME-VERIF . You can get a copy of YR-DB-RUNTIME-VERIF using the following command:

```
git clone https://github.com/yerothd/yr-db-runtime-verif
```

Creating a binary executable for State Diagram Mealy Machine (SDMM) specifications consists of the following elements:

- 'MariaDB' database connection configuration file: this file defines settings to connect to the system under test (SUT) application database; it is located in path: "\$YR-DB-RUNTIME-VERIF/YR-DB-RUNTIME-VERIF-GUI-ELEMENTS-SETUP/yr-db-runtime-verif-database-connection.properties".
A database connection to the SUT application database is required in order to check LTL property through the SDMM application library
YR_SD_RUNTIME_VERIF_LANG .
- Property configuration file: this file defines environment variables necessary for building a binary executable for the user; it is located in path: "\$YR-DB-RUNTIME-VERIF/\$USER_PROJECT/bin/configuration-properties.sh".

- "\$YR-DB-RUNTIME-VERIF/\$USER_PROJECT/sd-mealy-machine-specs": this directory contains user defined State Diagram Mealy Machine (SDMM) specifications to generate Corresponding runtime monitors within a single binary executable.
- Generate an executable for a user defined runtime monitor:

- execute following command in directory "\$YR-DB-RUNTIME-VERIF":

```
./YR-create-executable-for-user-SDMM.sh -d $USER_PROJECT
```

- modify the LTL verification code part within the generated source code files.

Then execute following command in directory "\$YR-DB-RUNTIME-VERIF":

```
./yr_db_runtime_verif_BUILD_DEBIAN_PACKAGE.sh
```

- uninstall YR-DB-RUNTIME-VERIF with following command in directory "\$YR-DB-RUNTIME-VERIF":

```
./yr_DB_RUNTIME_VERIF_uninstall.sh
```

- re-install YR-DB-RUNTIME-VERIF with following command in directory "\$YR-DB-RUNTIME-VERIF":

```
./yr_DB_RUNTIME_VERIF_INSTALL.SH
```

8 HOW TO START YR-DB-RUNTIME-VERIF


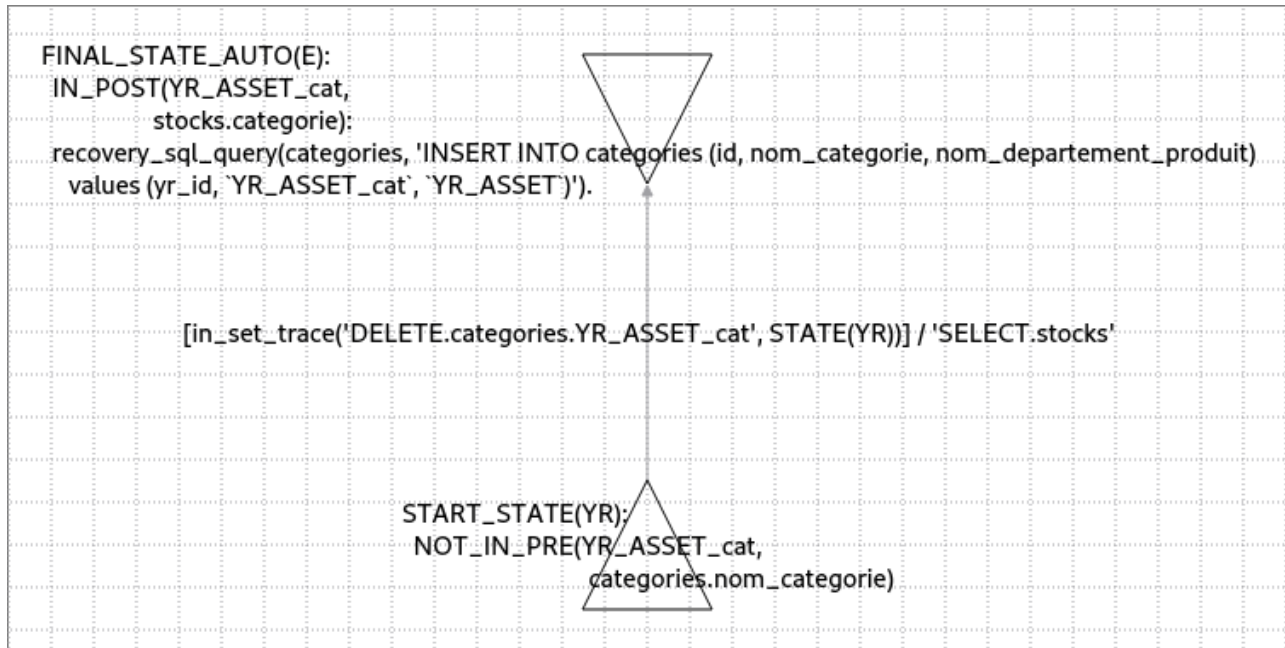
- The "ELF-x64" binary executable, in the source development directory is located in full path: "\$YR-DB-RUNTIME-VERIF/bin".
- The DEBIAN-LINUX icon () of YR-DB-RUNTIME-VERIF is located in "Applications" menu under section "Programming", and section "Accessories".
- The "ELF-x64" binary executable, after installation of the DEBIAN-LINUX package 'yr-db-runtime-verif.deb' is located in full path: "/opt/yr-db-runtime-verif/bin".

Figure 9: **SAMPLE sql recovery state diagram model in YEROTH_QVGE**

9 SQL QUERY Recovery execution on demand

A user can specify which SQL command query to execute whenever a System Under Test (SUT) lands in an accepting error state. This is done using keywords ending with "AUTO", used for meaning "AUTO RECOVERY FROM FAIL STATE":

1. `recovery_sql_query`
2. `END_STATE_AUTO`
3. `FINAL_STATE_AUTO`
4. `ERROR_STATE_AUTO`.

The use of an "AUTO" keyword shall be accompanied with a use of keyword `recovery_sql_query`, that specifies a SQL command query to run when landing in this fail error accepting state.

9.1 Automatic SQL Command Query Generation

YEROTH_QVGE implements an automatic SQL query generation strategy in case a user don't specify a SQL command query, since it could be leaved empty: Subsections 9.1.1, 9.1.2, 9.1.3, and 9.1.4 describe the strategy implemented.

9.1.1 ERROR ACCEPTING STATE for sdmm 1.

$$\frac{\text{not in_before}(YX, YY) \quad \text{ACTION}(V)}{\text{in_after}(DD, YR)}$$

9.1.2 RECOVERY 1.

$$\frac{\text{in_after}(DD, YR) \quad \text{ACTION}(\text{RECOVERY_ND})}{\text{not in_after}(DD, YR)}$$

9.1.3 RECOVERY 2 (Practical solution to be implemented in YR-DB-RUNTIME-VERIF.

$$\frac{\text{in_after}(DD, YR) \quad \text{ACTION}(\text{RECOVERY_D})}{\text{in_after}(YX, YY)}$$

9.1.4 Concrete RECOVERY 2 action.

$$\frac{\text{in_after}(YX, YY) \quad \text{insert_RECOVERY}(YX, YY)}{\text{in_before}(YX, YY)} \bullet$$

10 Formal Scientific and Engineering Project Description

Detailed formal scientific and engineering contributions of design and testing system YEROTH_QVGE can be found in *JOURNAL ARTICLE "Runtime Verification Of SQL Correctness Properties with YR-DB-RUNTIME-VERIF"* [Nou23].

11 Conclusion

The graphical drawing tool YEROTH_QVGE (Figure 4) costs only 3,000 EUROS. WE ONLY SUPPORT **DEBIAN-LINUX** (<https://www.debian.org>).

References

- [Nou23] Xavier N. Noundou. Runtime Verification Of SQL Correctness Properties with YR-DB-RUNTIME-VERIF. <https://zenodo.org/record/8381187>, October 2023.

Information Brochure of the Design and Testing System YEROTH_QVGE (YR_QVGE)

PROF. DR.–ING. DIPL.–INF. XAVIER NOUNDOU
CONTACT: yeroth.d@gmail.com

Table 1: EQUIVALENCES

scientific literature	engineering acronym
PRE	BEFORE
POST	AFTER
A TRACE	AN EVENT LOG
A FINAL STATE	AN ERROR STATE

Figure 1: A motivating example, as previous bug found in YEROTH-ERP-3.0.

$Q0 := \text{NOT_IN_BEFORE}(\text{YR_ASSET}, \text{department.department_name}).$

$Q1 := \text{IN_AFTER}(\text{YR_ASSET}, \text{stocks.department_name}).$

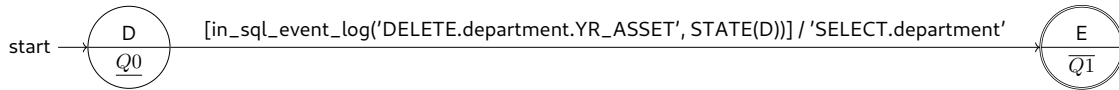


Figure 2: A SAMPLE state diagram mealy machine file.

```

1. yr_sd_mealy_automaton_spec yr_missing_department_NO_DELETE
2. {
3.   START_STATE(d):NOT_IN_BEFORE(YR_ASSET,department.department_name)
4.   ->[in_sql_event_log('DELETE.department.YR_ASSET',STATE(d))]/'SELECT.department'->
5.     ERROR_STATE(e):IN_AFTER(YR_ASSET,stocks.department_name).
6. }
  
```

Figure 3: A SCREENSHOT OF YEROTH_QVGE.

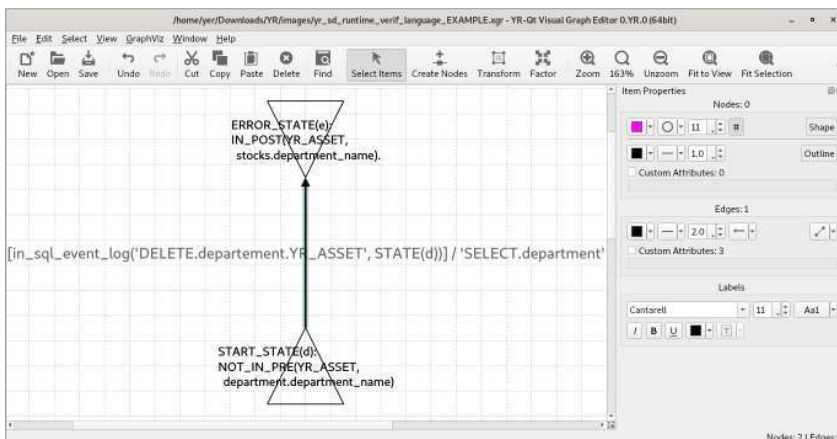


Figure 4: A SCREENSHOT OF YR-DB-RUNTIME-VERIF SQL EVENT LOG.

timestamp	sql event log	source	target	changed qty
06-10-2023	SELECT * FROM stocks WHERE department = 'YR_ASSET'	YR-DB-RUNTIME-VERIF	YR-DB-RUNTIME-VERIF	1
06-15-2024	SELECT * FROM stocks WHERE department = 'YR_ASSET'	YR-DB-RUNTIME-VERIF	YR-DB-RUNTIME-VERIF	1

timestamp	SQL recovered executed query
06-10-2023	SELECT * FROM stocks WHERE department = 'YR_ASSET'

resource file	line number
src/windows/stocks/verif/erp_stocks_window.cpp	2149

before pre-condition on source state	after post-condition on target state
NOT_IN_BEFORE(YR_ASSET, department.department_name)	IN_AFTER(YR_ASSET, stocks.department_name)

evaluated guarded condition expression	value	previous state	accepting state	is error state
in sql event log[SQL event log] ASSET = 'YR_ASSET'	true	0	0	Yes

1 Developer Biography



Figure 5: Portrait of XAVIER.

PROF. DR.-ING. DIPL.-INF. XAVIER NOUNDOU is a CHRISTIAN BY FAITH, Cameroonian, born on September 16 1983 in DOUALA (LITTORAL region, CAMEROON). Xavier has a "Diplom-Informatiker (Dipl.-Inf.)" qualification from the **University of Bremen, Bremen, Bremen, GERMANY** (May 25, 2007). XAVIER NOUNDOU IS A **PHILOSOPHIAE DOCTOR (PH.D.)** from **THE UNIVERSITY OF WATERLOO (ON, CANADA); DECEMBER 20, 2011 !**

PROF. DR.-ING. DIPL.-INF. XAVIER NOUNDOU has worked together with **PROF. DR. RER. NAT. HABIL. JAN PELESKA**, at AGBS-University of Bremen, GERMANY; and 2 years later at WatForm-University of Waterloo, ON, Canada, with **PATRICK LAM, PH.D. (MIT, BOSTON, MA, USA), P.ENG. (Ontario, CANADA)**.

Xavier could successfully work with **DR. FRANK TIP** at The University of Waterloo (Waterloo, ON, Canada) on his first JAVA dynamic program analysis.

Xavier also had the great opportunity through **DR. MARCEL MITRAN** and **PATRICK LAM, PH.D., P.ENG.**; to work as a graduate intern in Markham (Toronto, ON, CANADA) at IBM TORONTO SOFTWARE LABORATORY; in the JAVA-J9 Just-In-Time Compiler Optimization Team, together with **VIJAY SUNDARESAN, M.Sc (McGill University, QC, Canada)**.

Xavier has following academic and professional engineering research contributions:

1. 'Statistical test case generation for reactive systems' at RTT-MBT at VERIFIED SYSTEMS INTERNATIONAL GmbH (<https://www.verified.de>).
2. 'Context-Sensitive Staged Static Taint Analysis For C using LLVM'
 1. source code in C++:
<https://github.com/sazzad114/saint>
 2. full text: <https://zenodo.org/record/8051293>
3. 'YEROTH-ERP-3.0':
 1. source code in C++:

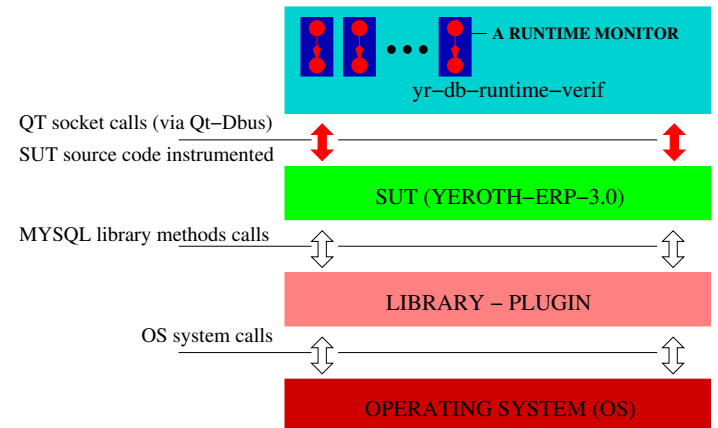
¹https://github.com/yerothd/yr_sd_runtime_verif_lang

²<https://github.com/yerothd/yr-db-runtime-verif>

- a. YEROTH-ERP-3.0:
<https://github.com/yerothd/yeroth-erp-3-0>
 - b. YEROTH-ERP-3.0 SYSTEM DAEMON:
<https://github.com/yerothd/yeroth-erp-3-0-system-daemon>
2. full text (ongoing publication):
<https://zenodo.org/record/8052724>

2 Introduction

Figure 6: SOFTWARE ARCHITECTURE OF YR-DB-RUNTIME-VERIF.



YEROTH_QVGE is a CASE (Computer-Aided Software Engineering) design tool to generate "domain-specific language (DSL) YR_SD_RUNTIME_VERIF_LANG¹" files, to be inputted into the "compiler YR_SD_RUNTIME_VERIF_LANG_COMP", so to generate C++ files for the runtime verifier tester "YR-DB-RUNTIME-VERIF²" that allows for manual verification of SQL correctness properties of Graphical User Interface (GUI) software.

YR-DB-RUNTIME-VERIF inputs SQL correctness properties expressed using the formalism state diagram mealy machine (YR_SD_RUNTIME_VERIF_LANG). Figure 6 illustrates a software system architecture of YR-DB-RUNTIME-VERIF , together with the monitored program under analysis. The Free Open Source Code Software (FOSS) tool-chain of development testing is located as follows for free, EXCEPT for "YEROTH_QVGE " that is a Closed Source Code Software (CSCS):

- COMPILER (i.e.: YR_SD_RUNTIME_VERIF_LANG_COMP): https://github.com/yerothd/yr_sd_runtime_verif_lang
- RUNTIME VERIFIER TESTER (i.e.: YR-DB-RUNTIME-VERIF): <https://github.com/yerothd/yr-db-runtime-verif>
- state diagram mealy machine UNIT TESTS CODE (i.e.: YR_SD_RUNTIME_VERIF_UNIT_TESTS): https://github.com/yerothd/yr_sd_runtime_verif_UNIT_TESTS
- state diagram mealy machine (i.e.: YR_SD_RUNTIME_VERIF_LANG): https://github.com/yerothd/yr_sd_runtime_verif

3 YEROTH_QVGE (YR_QVGE) Project Dependency

Table 2: YEROTH_QVGE Design and Testing System Dependencies

PROJECT	Required Library
1) YR_SD_RUNTIME_VERIF_LANG	
2) YR_SD_RUNTIME_VERIF_LANG_COMP	1)
3) YR_SD_RUNTIME_VERIF_UNIT_TESTS	1)
4) YR-DB-RUNTIME-VERIF	2)

Table 2 illustrates for each library project, which others it depends on.

Figure 7: YEROTH_QVGE software library dependencies.

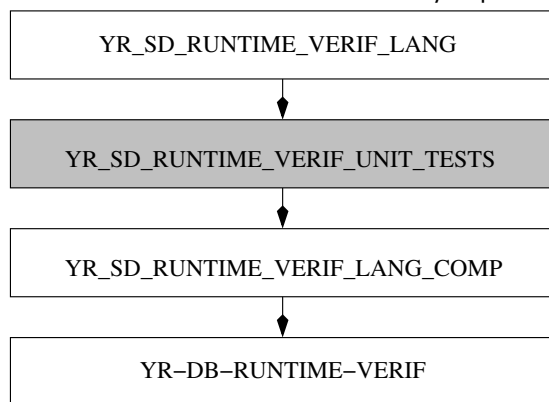


Figure 7 show a diagram overview of the presentation in Table 2. The step of the unit tests is colored in gray because it is only for developers of YEROTH_QVGE intended.

4 Potential Uses of YEROTH_QVGE

YEROTH_QVGE (YR_QVGE) could be used for the following automatic generation, analysis, verification, and validation tasks:

1. Automatic generation of runtime monitoring module program to prove whether a test procedure, automated, or not, is correct with regards to a test and / or design STATE DIAGRAM MEALY MACHINE.

In effect, let the test execution be runtime monitored to watch whether accepting error states would be found.

For instance, Junit testing environment could automatically integrate an automatically generated runtime monitor infrastructure for unit testing.

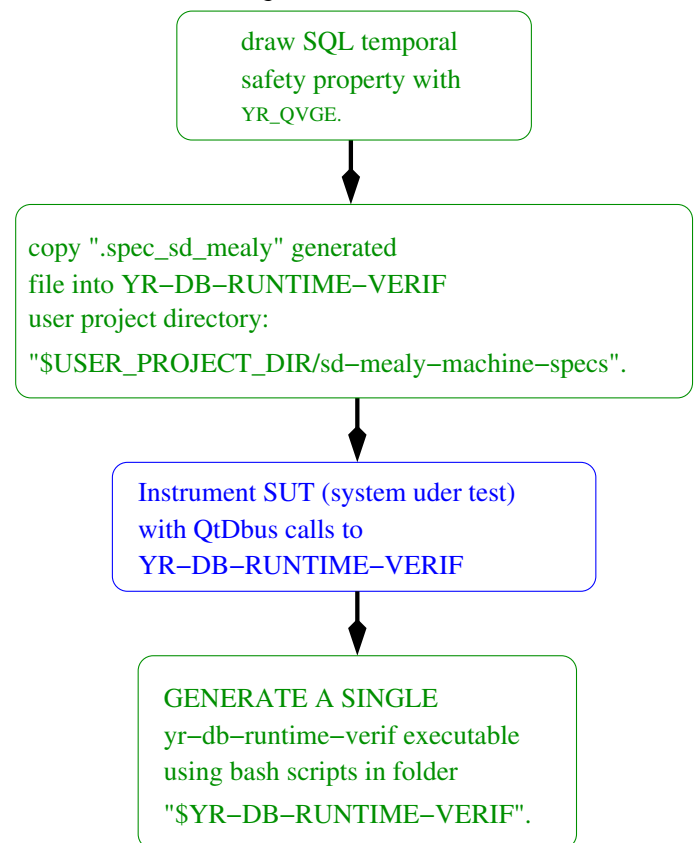
2. Automatic generation of runtime monitoring module program for any software that can emit Dbus messages.

Such runtime monitoring modules are for interest for special LTL model checking properties that cannot get a definite answer through use of a conventional model checker.

3. Software design properties with SQL
4. Software design properties including event sequences over different layers of software system architecture
5. Class diagram with sequence diagram.

5 Advantages of YEROTH_QVGE

Figure 8: Workflow.



A sample state diagram mealy machine is shown in Figure 2.

WITH manual drawing of SQL CORRECTNESS PROPERTY MODEL, you are freed from manually writing "state diagram mealy machine text files" that could be tedious and lengthy. Also, editing state diagram mealy machine files manually could be more error-prone than letting a compiler (YR_SD_RUNTIME_VERIF_LANG_COMP) do it for you.

6 Conclusion

YEROTH_QVGE costs only 3,000 EUROS. WE ONLY SUPPORT DEBIAN-LINUX (<https://www.debian.org>).

Index

state diagram mealy machine, [2](#)

CASE (Computer-Aided Software Engineering), [2](#)

domain-specific language (DSL), [2](#)

runtime verifier tester, [2](#)

SQL correctness properties, [2](#)

YEROTH-ERP-3.0 SOFTWARE SYSTEM ARCHITECTURE

PROF. DR.-ING. DIPL.-INF. XAVIER NOUNDOU

This document describes the thick client software system architecture of YEROTH-ERP-3.0. This document also explains the reasons for which we chose to design and implement YEROTH-ERP-3.0 as a thick client software system, as opposed to currently more popular webbrowser based software system.

This document further demonstrates the superiority, in terms of simplicity, speed, maintenance, and low costs of development of thick client software system architectures over webbrowser based software system architectures !

1. YEROTH-ERP-3.0, source code (no executable binary):
<https://github.com/yerothd/yeroth-erp-3-0>
2. YEROTH-ERP-3.0-SYSTEM-DAEMON, source code (no executable binary):
<https://github.com/yerothd/yeroth-erp-3-0-system-daemon>
3. YR-DB-RUNTIME-VERIF:
<https://github.com/yerothd/yr-db-runtime-verif>
4. YR_SD_RUNTIME_VERIF:
https://github.com/yerothd/yr_sd_runtime_verif

Version: November 7, 2023.

Contents

Contents	3
List of Figures	5
Listings	7
List of Tables	9
1 Introduction	11
1.1 Motivation	11
1.1.1 Application Domain	12
1.1.2 Implementation Technology	12
1.1.3 Software System Current Metrics	14
1.2 Overview	14
2 Thick Client VS. Webbrowser based Software System Architecture	15
2.1 Thick client: 2 layers logical software architecture	15
2.2 Webbrowser based: at least 4 layers logical software architecture	15
2.3 Tabular Comparison Between Thick-Client And Webbrowser based Architecture	16
3 The Thick Client Software System Architecture of YEROTH–ERP–3.0	17
3.1 Business and user interface code deployment	17
3.2 Databases	17
3.3 NUMBER OF LOGICAL SOFTWARE SYSTEM LAYERS: E.G. Sample technical configurations	17
3.3.1 Sample 2 computers store	18
3.3.2 Sample decentralized multi sites supermarket	18
4 Conclusion	19
5 Bibliography	21
Index	23
A Presentation Documents of open source software system YEROTH–ERP–3.0	25

List of Figures

1.1	Business manager's main window	11
2.1	2 layers logical architecture of thick client software system (Image copied from [sec20]).	15
2.2	4 layers logical architecture of webbrowser based software system (Image copied from [KM06]).	15
3.1	Sample 2 computers store.	18
3.2	Sample decentralized multi sites supermarket.	18

Listings

List of Tables

1.1	COMPARISON TABLE BETWEEN C++ [Ste90], JAVASCRIPT [Fla20], AND JAVA [AGH00]. JAVASCRIPT and JAVA only have virtual machines that are in FACT REACTIVE SYSTEMS!	12
1.2	YEROTH–ERP–3.0 RELEVANT SOFTWARE SYSTEM METRICS	14
2.1	Thick client application VS Webbrowser based application.	16
3.1	Thick client application VS Webbrowser based application.	17
4.1	YEROTH–ERP–3.0 VS. Odoo.	19

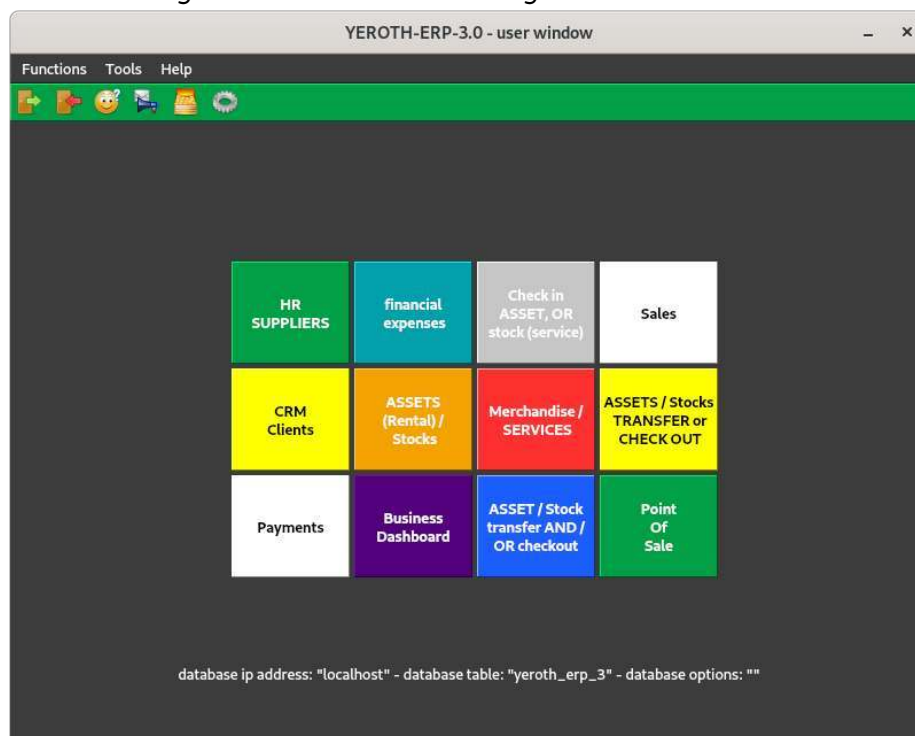
Chapter 1

Introduction

This introduction motivates why I created YEROTH-ERP-3.0, and why it uses the best software programming language AND METHODOLOGY (OOP: OBJECT-ORIENTED PROGRAMMING) of its time !

1.1 Motivation

Figure 1.1: Business manager's main window



YEROTH-ERP-3.0 [NOU22, NOU21a, NOU21b] is an **Enterprise Resource Planing (ERP)** software system that aims 'effectiveness' and 'simplicity', compared to other high ranked ERP software systems (e.g.: 'Sage Gescom i7', 'SAP Business One', 'Odoo', etc.).

YEROTH-ERP-3.0 is implemented by me with THE USER INTERFACE LIBRARY QT [Ltd22]. Our goal when designing and implementing this ERP software system is morefolds:

- 1) RENDER ENTERPRISE RESOURCE PLANING (ERP) SOFTWARE SYSTEM USAGE AND MANIPULATION AS EASY AS READING A LEISURE OR RECREATIONAL BOOK SO TO REDUCE USER STRAIN SIZE !
- 2) CONTRIBUTE TO REDUCE POVERTY BY IMPROVING SOFTWARE SYSTEM TECHNOLOGY FOR ENTERPRISE RESOURCE PLANING (ERP).

1.1.1 Application Domain

YEROTH–ERP–3.0 can be used by any managerial organization, WHETHER governmental, OR NON GOVERNMENTAL (N.G.O) !

YEROTH–ERP–3.0 is aimed at being simpler in usage and manipulation than older top ranked ERP software systems (e.g.: 'Sage Gescom i7', 'SAP Business One', Odoo, etc.) !

A DETAILED COMPARISON OF OCCURS IN PAPER [NOU21a]!

Applications domains of YEROTH–ERP–3.0 are for instance:

1. engineering offices;
2. insurance companies;
3. attorney offices;
4. etc.

1.1.2 Implementation Technology

Table 1.1: COMPARISON TABLE BETWEEN C++ [Ste90], JAVASCRIPT [Fla20], AND JAVA [AGH00]. JAVASCRIPT and JAVA only have virtual machines that are in FACT RE-ACTIVE SYSTEMS !

FEATURES	C++	Javascript	Java
1. PROGRAMMING LANGUAGE PHILOSOPHY	object-oriented	object-oriented	object-oriented
2. MACROS	✓	✓	
3. MULTIPLE INHERITANCE	✓		
4. support INTERFACES	✓	✓	✓
5. STATIC COMPILATION	✓		
6. VM ¹ INTERPRETATION AND MANAGEMENT		✓	✓
7. UNIT TESTING FRAMEWORK	✓	✓	✓
8. NETWORK CAPABLE	✓	✓	✓
9. CROSS PLATFORM PORTABLE	✓	✓	✓
10. include FILE LIBRARY CAPABILITY	✓		
11. MULTI THREADING	✓		✓
12. WYSIWYG GUI DESIGN TOOL	✓		✓
13. VIRTUAL MACHINE IS A REACTIVE SYSTEM		✓	✓

We chose to design and implement YEROTH–ERP–3.0 as a thick client software system because of the following reasons:

1. The implementation language C++ offers much flexibility:

1. *MULTIPLE INHERITANCE:*

It allows developers to abstract as much as possible business code upwards, away from downwards implementation classes. For instance, in YEROTH–ERP–3.0, GUI Qt windows inherits for instance *search filtering feature*, and *print capability* from 2 different classes.

Print capability couldn't be inherited from the same class where search filtering is abstracted and partially implemented (interface in JAVA for instance doesn't allow any method body code), because it works in its pure abstract class (C++ class with at least 1 empty method body), together with feature **database column filtering for viewing and printing**.

The drawback of the multiple inheritance in C++ is: it sometimes can be very difficult to build it using "gcc (g++) [GCC]"!

2. *MACROS:*

They enable developers to create TEXT TEMPLATE in their code.

For instance, I use macros in some parts of my code to reduce execution time and stack activation records size for method or function calls in YEROTH–ERP–3.0!

2. The availability of 'WHAT YOU SEE IS WHAT YOU GET' (WYSIWYG) tools for fast and useful user interface design (e.g.: Qt designer [Com20], miniStudio (vxWorks) [WEI20], etc.)
3. The low number of logical software system architecture layer (i.e.: 2.) involved with the use of a thick client software system architecture, as opposed to a webbrowser based software system (i.e.: 4, *client user interface*, *presentation layer*, *business logic*, and *data (DBMS)*).

FURTHERMORE, TABLE 1.1 ILLUSTRATES A FEATURE COMPARISON TABLE BETWEEN object oriented languages C++, JAVASCRIPT, and JAVA!

1.1.3 Software System Current Metrics

Table 1.2: YEROTH–ERP–3.0 RELEVANT SOFTWARE SYSTEM METRICS

Software System Metric	Value
User Interface (windows, dialog) number	60
MariaDB SQL table number	39
MariaDB SQL table column number	320
Latex template for PDF printing	75
Source lines of code (SLOC)	330,000

1.2 Overview

This pamphlet is structured as follows:

1. Chapter 1 motivates why I created YEROTH–ERP–3.0, and why it uses the best software programming language AND METHODOLOGY (OOP: OBJECT–ORIENTED PROGRAMMING) of its time !
2. Chapter 2 tabular evaluates why thick client are BETTER THAN webbrowser based software system architectures !
3. Chapter 3 explains why YEROTH–ERP–3.0 is modular in its uses, and fits any industrial setting.
4. Chapter 4 explains why YEROTH–ERP–3.0 uses the BEST SOFTWARE TECHNOLOGY IN TERMS OF SOFTWARE SYSTEM ARCHITECTURE !

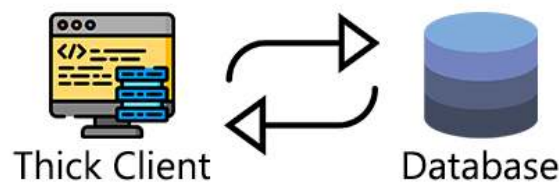
Chapter 2

Thick Client VS. Webbrowser based Software System Architecture

This comparison chapter tabular evaluates why thick client are BETTER THAN webbrowser based software system architectures !

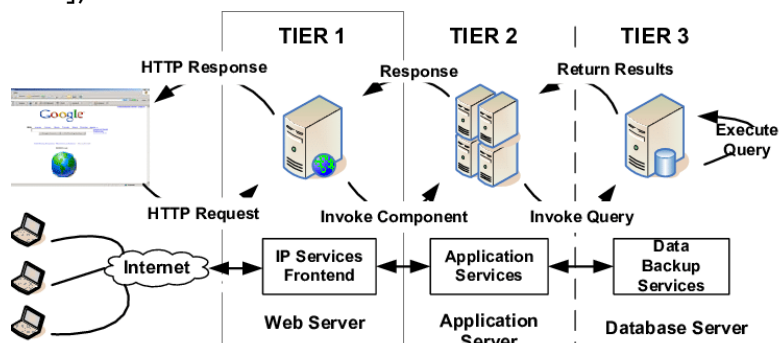
2.1 Thick client: 2 layers logical software architecture

Figure 2.1: 2 layers logical architecture of thick client software system (Image copied from [sec20]).



2.2 Webbrowser based: at least 4 layers logical software architecture

Figure 2.2: 4 layers logical architecture of webbrowser based software system (Image copied from [KM06]).



2.3 Tabular Comparison Between Thick-Client And Web-browser based Architecture

Table 2.1: Thick client application VS Webbrowser based application.

	Thick client application ✓	Webbrowser based application
business code	user interface	application server
co-related software systems	1 (DBMS)	at least 3 (DBMS, web / application server)
number of logical layers	2 (client and data)	4 (client, presentation, logic, and data)
rapid prototyping (WYSIWYG tools)	yes	very limited
software security vulnerability	low (1 programming language)	VERY high (several programming languages)
user interface	all computers (GUI with BUSINESS CODE)	all computers (web-browser)

Table 2.1 compares thick client software systems against webbrowser based software systems.

Table 2.1 ALSO ILLUSTRATES ADVANTAGES of thick client software system architecture over webbrowser based software system architecture !

The common argument for webbrowser based software system architecture is you update the business code just at 1 place: *the application server* !

I argue that thick client architecture IS JUST AS WELL BEST UPDATED AT 1 PLACE: *the user's computer*.

FOR INSTANCE, UPDATE OR UPGRADE OF ENTERPRISE SOFTWARE SYSTEM WEBBROWSER BASED REQUIRES ALMOST AT LEAST 1 24 HOURS SHUTDOWN OF ALL INVOLVED WEB (apache tomcat, etc.) APPLICATIONS SERVERS.

The issue of automatic software upgrade in a computer network is best solved by the 'apt upgrade software system of Debian-Linux', as an example !

Chapter 3

The Thick Client Software System Architecture of YEROTH–ERP–3.0

This chapter explains why YEROTH–ERP–3.0 is modular in its uses, and fits any industrial setting !

Table 3.1: Thick client application VS Webbrowser based application.

	Thick client application ✓	Webbrowser based application
business code	user interface	application server
co-related software systems	1 (DBMS)	at least 3 (DBMS, web / application server)
number of logical layers	2 (client and data)	4 (client, presentation, logic, and data)
rapid prototyping (WYSIWYG tools)	yes	very limited
software security vulnerability	low (1 programming language)	high (several programming languages)
user interface	all computers (GUI with BUSINESS CODE)	all computers (web-browser)

3.1 Business and user interface code deployment

Table 3.1 depicts the issue of business and user interface code deployment on all computers participating in the functioning of YEROTH–ERP–3.0, as a software system for a user.

We tackle the problem of automatic deployment of business and user interface code on all user computers by using the 'apt upgrade' software system on 'Debian-Linux [DEB22]'.

3.2 Databases

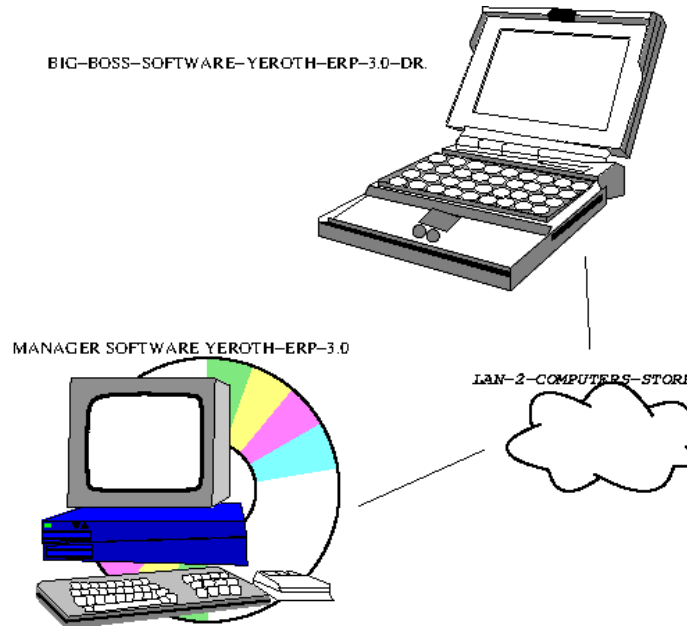
DBMS MySQL [Mar22] is used for storing and managing huge data across globe.

3.3 NUMBER OF LOGICAL SOFTWARE SYSTEM LAYERS: E.G. Sample technical configurations

This section illustrates 2 different possible computer network configurations that could prevail in industry.

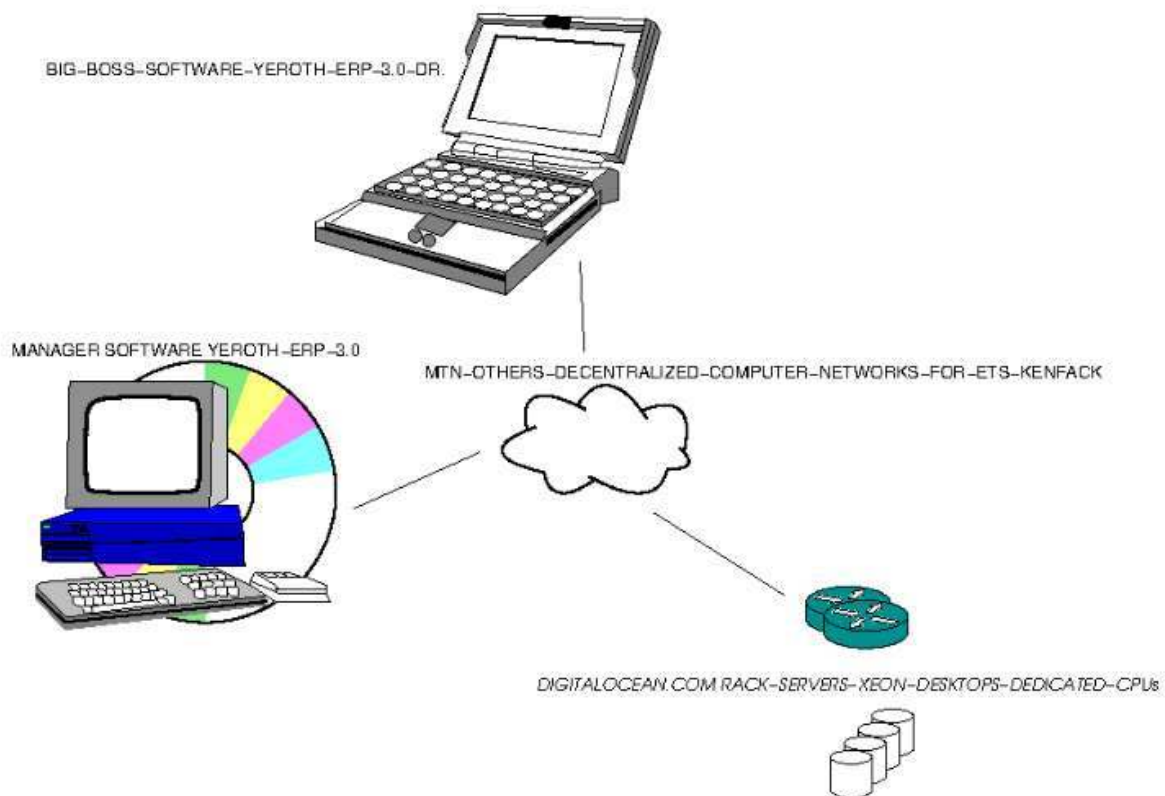
3.3.1 Sample 2 computers store

Figure 3.1: Sample 2 computers store.



3.3.2 Sample decentralized multi sites supermarket

Figure 3.2: Sample decentralized multi sites supermarket.



Chapter 4

Conclusion

This conclusion explains why YEROTH–ERP–3.0 uses the BEST SOFTWARE TECHNOLOGY IN TERMS OF SOFTWARE SYSTEM ARCHITECTURE !

Table 4.1: YEROTH–ERP–3.0 VS. Odoo.

	YEROTH–ERP–3.0	Odoo
libraries & programs	lxqt-sudo, etc.	python-lxml, etc.
user interface code TOOLS WYSIWYG	QT–DESIGNER	(CUSTOM BUILD) FRAMEWORKS
business code	C++	Python, JavaScript
Database Management Server (DBMS)	MySQL	PostgreSQL
web server		Werkzeug

YEROTH–ERP–3.0 has a thick client software system architecture because we found thick client software system architectures simpler than webbrowser based software system architectures.

Thick client software system architectures is simpler because it requires less layers in its logical (or physical) software system architecture, and is easier to develop and maintain as a software system application.

Table 2.1 illustrates a thick client software system is SUPERIOR IN TERMS OF TOOLS FOR MAINTENANCE AND DEVELOPMENT than a webbrowser based software system !

A webbrowser based software system architecture has more drawbacks as follows:

1. it requires at least 2 other software systems, *apart from the ones normally required by developed software system itself, for instance libraries (e.g.: Log4j), to fully operate (e.g.: web server, application server, etc.).*

Table 4.1 depicts this situation in the light of the open source ERP software system Odoo.

Accordingly, a thick client software system doesn't require any running and managing infrastructure such as for example an application server !

2. A webbrowser based software system requires at least 4 layers in its logical system architecture (e.g.: client, presentation, logic, and data layers).

Accordingly, a thick client software system only requires at least 2 layers !

3. A webbrowser based software system potentially entails more software security vulnerabilities because its implementation requires the use of at least 2 different programming languages, and frameworks in combination.

Accordingly, a thick client software system needs only the use of 1 homogeneous software programming language !

Chapter 5

Bibliography

- [AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison Wesley Longman Publishing Co., Inc., USA, 3rd edition, 2000.
- [Com20] The Qt Company. Qt Designer Manual. <https://doc.qt.io/qt-5/qtdesigner-manual.html>, 2020. Last accessed on September 4, 2020 at 15:21.
- [DEB22] DEBIAN. Debian – The Universal Operating System. <https://www.debian.org>, 2022. ACCESSED LAST TIME on JUNE 12, 2022 at 12:10.
- [Fla20] David Flanagan. *JavaScript: The Definitive Guide: Master the World’s Most-Used Programming Language 7th Edition*. O’Reilly, 2020.
- [GCC] THE COMPILER SUITE GCC. THE GCC (G++) COMPILER SUITE. <https://www.gnu.org/>. Last accessed on December 29, 2020 at 12:00.
- [KM06] Taeho Kgil and Trevor Mudge. Flashcache: A nand flash memory file cache for low power web servers. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES ’06, page 103–112, New York, NY, USA, 2006. Association for Computing Machinery.
- [Ltd22] The Qt Company Ltd. Qt is a C++ toolkit for cross-platform application development. <https://www.qt.io>, 2022. ACCESSED LAST TIME on JUNE 12, 2022 at 18:00.
- [Mar22] MariaDB.org. MariaDB Foundation - MariaDB.org. <https://www.mariadb.org>, 2022. ACCESSED LAST TIME on JUNE 24, 2022 at 12:20.
- [NOU21a] XAVIER NOUNDOU. GROUPED PRESENTATIONS DOCUMENTS OF YEROTH–PGI–3.0. https://archive.org/download/yeroth-erp-3-0-info-english_202104/yeroth-erp-3-0-info-english.pdf, 2021. ACCESSED LAST TIME ON june 17, 2022 at 08:40.
- [NOU21b] XAVIER NOUNDOU. INSTALLATION GUIDE FOR ERP SOFTWARE SYSTEM YEROTH–ERP–3.0. <https://archive.org/download/yeroth-erp-3-0-installation-guide-standalone/yeroth-erp-3-0-installation-guide-standalone.pdf>, 2021. ACCESSED LAST TIME ON june 17, 2022 at 08:20.

- [NOU22] XAVIER NOUNDOU. YEROTH-ERP-PGI-3.0 DOCTORAL COMPENDIUM. https://archive.org/download/yeroth-erp-pgi-compendium_202206/JH_NISSI_ERP_PGI_COMPENDIUM.pdf, 2022. ACCESSED LAST TIME ON June 22, 2022 at 12:00.
- [sec20] securityboulevard.com. Thick Client Penetration Testing Methodology. <https://securityboulevard.com/2020/02/thick-client-penetration-testing-methodology/>, 2020. Last accessed on September 4, 2020 at 15:21.
- [Ste90] W. Richard Stevens. *UNIX Network Programming*. Prentice-Hall, Inc., USA, 1990.
- [WEI20] Yongming WEI. miniStudio User's Guide. <https://www.minigui.net/en/ministudio>, 2020. Last accessed on September 4, 2020 at 15:21.

Index

1–page presentation of YEROTH–ERP–3.0, [26](#)
2 layers logical architecture of thick client software system, [15](#)
2–pages presentation of YEROTH–ERP–3.0, [31](#)
4 layers logical architecture of webbrowser based software system, [15](#)
YEROTH–ERP–3.0 VS. Odoo webbrowser based software system, [19](#)

comparison of YEROTH–ERP–3.0 against others, [27](#)
comparison table between thick client and webbrowser based software system, [16](#)

motivation for creating YEROTH–ERP–3.0, [11](#)

point of sale proposed hardware, [28](#)

Sample 2–computers store, [18](#)
sample decentralized multi sites supermarket, [18](#)
STRUCTURE OF THIS PAPER, [14](#)

Appendix A

Presentation Documents of open source software system YEROTH–ERP–3.0

YEROTH–ERP–3.0 Software System Product Sheet

YEROTH–ERP–3.0 is an **ERP software system** with 6 **user roles, and types**:

1. « Administrator »
2. « Business manager »
3. « Cashier »
4. « Seller »
5. « ASSET – stock manager »
6. « Storekeeper ».

YEROTH–ERP–3.0 **features**:

1. alerts over stock quantity, and, time period
2. business dashboard
3. HR (human resources), customer relationship management (CRM), budget line management
4. sale management (e.g. point of sale)
5. ASSET – stock management (e.g. check in)
6. user, and role administration
7. wildchar searches with character %.

YEROTH–ERP–3.0 **is**:

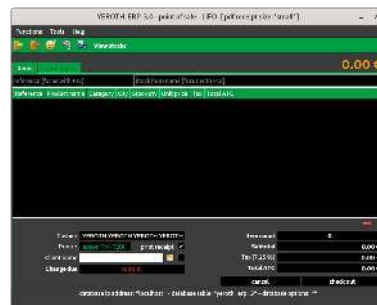
1. easier, and, intuitiver, in its use
2. lighter, and, faster, in memory usage
3. multi sites.

YEROTH–ERP–3.0's runtime memory usage test is realized using run–time software analysis tool **valgrind**.

GENERAL SOURCE CODE QUALITY CONTROL is realized with compile–time code analysis tool **Cppcheck**.



Business manager's main window



Cashier's main window

OPERATIONS

Point of Sale Hardware

- ✓ Barcode scanner
- ✓ Thermal printer, etc.



Database Management Systems

- ✓ MySQL



Operating Systems

- ✓ Debian–Linux



Advantages of YEROTH–ERP–3.0 Compared to other top ERP Software Systems

YEROTH–ERP–3.0 is a very easy to use ERP (Enterprise Resource Planing) software system because of its characteristics:

1. separate views for each user role
2. complete and fundamental training in 5 days
3. easy to use graphical user interface (GUI)
4. no college or university training needed
5. no formal business training needed
6. no financial accounting training needed
7. no internet connection needed.



Stock listing window

Table 1 pictures the 'effectiveness' and 'simplicity' of YEROTH–ERP–3.0, compared to other top ERP software systems "Sage Gescom i7", and "SAP Business One".

	YEROTH–ERP–3.0	Sage Gescom i7	SAP Business One
separate views per user role	YES	YES	YES
complete training (or solution)	at least 2 weeks	at least 2 months	at least 3 months
difficulty in navigation	easy	very difficult	very difficult
usage language in software	easy everyday English	simple	technical
financial accounting knowledge	no	no	useful
advanced marketing knowledge	no	useful	useful
internet connection	optional	optional	optional

Table 1: Comparison between YEROTH–ERP–3.0 and 2 top tier–1 full featured ERP software systems

OPERATIONS

Point of Sale Hardware

- ✓ Barcode scanner
- ✓ Thermal printer, etc.



Database Management Systems

- ✓ MySQL



Operating Systems

- ✓ Debian–Linux



YEROTH–ERP–3.0 Point Of Sale Recommended Hardware

1 Barcode Scanner

We recommend, but not exclusively, the use of barcode scanner: " **Xfox FJ–5 USB Plug and Play Automatic Barcode Scanner** " (approx. 17€).



Barcode Scanner

2 Thermal Printer

ALL EPSON THERMAL PRINTERS !

We recommend, but not exclusively, the use of thermal printer: " **Epson TM T20ii Point of Sale Thermal Printer** " (approx. 100€).



Thermal Printer

3 Cash Drawer

We recommend, but not exclusively, the use of cash drawer: " **HP QT457AT** " (approx. 90€).



Cash Drawer

4 Touch Screen Monitor

We recommend, but not exclusively, the use of touch screen monitor: " **ASUS 15.6" LCD Monitor (VT168H)** " (approx. 155€).



Touchscreen Monitor

5 Computer

We recommend, but not exclusively, the use of desktop computer: " **Lenovo Thinkcentre M720 Small Form Factor (SFF)** " (approx. 450€).



Computer

6 MULTI USER TERMINAL Computer (*)

We recommend, but not exclusively, the use of multi user terminal computer: " **NC–300 Multi User Terminal Computer** " (approx. 112€).



Multi User Terminal Computer

✓ **PECULIARITIES IN CERTAIN COUNTRIES AND/OR REGIONS** USAGE OF POINT–OF–SALE (THERMAL) PRINTERS REQUIRES ALSO ACQUISITION OF A GOVERNMENT–SOLD DEVICE FOR RECORDING AND CERTIFYING ALL FINANCIAL TRANSACTIONS BETWEEN THE THERMAL PRINTERS AND/OR YEROTH–ERP–3.0 (e.g.: GERMANY, CANADA, etc.).

Information Brochure of the ERP software system YEROTH–ERP–3.0

Dr.–Ing. Xavier Noubissi Noundou

Tasks	« Business manager »	« Seller »	« ASSET – stock manager »	« Storekeeper »	« Cashier »
create a department	✓				
insert 1 ASSET, stock, or service	✓	✓ (SERVICE)	✓ (ASSET / STOCK)		
delete ASSET, stock	✓				
view ASSET, stock	✓	✓	✓	✓	✓
modify ASSET, stock	✓		✓		
transfer ASSET, stock	✓		✓	✓	
check–out ASSET, stock	✓		✓	✓	
modify ASSET, stock management strategy (e.g.: « FIFO », etc.)	✓	✓ (NO PERMANENT)	✓ (NO PERMANENT)		
point of sale	✓	✓			✓
view ASSET, stock transfers	✓		✓	✓	
purchase management	✓	✓	✓ (PARTIAL)		
supplier management	✓	✓			
customer relationship management (CRM)	✓	✓ (PARTIAL)			
business dashboard	✓				
sale return	✓				
view sales information	✓	✓ (SELF)			

Table 1: YEROTH–ERP–3.0 functions–tasks, and associated users–roles.

1 Developer Biography



Figure 1: Portrait of Dr.–Ing. XAVIER.

Dr.–Ing. Xavier Noubissi Noundou is a CHRISTIAN BY FAITH, Cameroonian, born on September 16 1983 in DOUALA (LITTORAL region, CAMEROON).

Xavier has a “Diplom–Informatiker (Dipl.–Inf.)” qualification from the **University of Bremen, Bremen, Bremen, GERMANY** (May 25, 2007).

XAVIER NOUMBISSI NOUNDOU IS A PHILOSOPHIAE DOCTOR ABD (PH.D. ABD) from **THE UNIVERSITY OF WATERLOO (ON, CANADA); DECEMBER 20, 2011 !**

Xavier has following academic research and professional engineering contributions:

1. ‘Context-Sensitive Staged Static Taint Analysis For C using LLVM’

1. source code in C++:

<http://github.com/sazzad114/saint>
<http://archive.org/download/>

2. full text: [yeroth-saint-2021-MARCH-01/YEROTH-SAINT-2021-MARCH-01.pdf](http://archive.org/download/yeroth-saint-2021-MARCH-01/YEROTH-SAINT-2021-MARCH-01.pdf)

2. ‘YEROTH-ERP-3.0’:

1. source code in C++:

- a. YEROTH–ERP–3.0:

http://drive.google.com/file/d/1-JJke20aa_fuIj3twWsBkj3-KRxgZL4q/view?usp=share_link

- b. YEROTH–ERP–3.0 SYSTEM DAEMON:

http://drive.google.com/file/d/1kn5_1KvWPkFD0VxA8eGvk-kuI08L-_T/view?usp=share_link

2. full text (ongoing publication):

http://archive.org/download/yeroth-erp-pgi-compendium_202206/YEROTH_ERP_PGI_COMPENDIUM.pdf

2 Introduction

YEROTH–ERP–3.0 is an **Enterprise Resource Planing** (ERP) software system.

Users of YEROTH–ERP–3.0 could have the following roles:

1. « Administrator »
2. « Business manager »
3. « Cashier »
4. « Seller »
5. « ASSET – stock manager »
6. « Storekeeper ».

YEROTH–ERP–3.0 allows for business management tasks (listed in Table 1), depending on user role, as follows:

1. create a department (e.g.: finance, asset, stock, etc.)
2. manage clients, and human resources, and suppliers
3. manage enterprise asset (e.g.: cars, etc.)
4. manage financial expenses
5. manage inventory stock
6. manage sales
7. view business dashboards (across sites).

3 Potential Usages of YEROTH–ERP–3.0

YEROTH–ERP–3.0 potential usages are:

1. STOCK AND TRADE EXCHANGES MARKET PLACE
2. ENTERPRISE RESOURCE AND PLANING SOFTWARE for supermarkets and commercial stores
3. i.e. ANY NON GOVERNMENTAL ORGANIZATION, or GOVERNMENTAL ORGANIZATION.

4 Advantages of YEROTH–ERP–3.0

1. YEROTH–ERP–3.0 is 100% stable
2. YEROTH–ERP–3.0 has an alert system with two types of alerts: alerts based on stock–quantity, and time–period alerts
3. users have the choice between small size receipts, and, bigger size receipts ("A4")
4. YEROTH–ERP–3.0 runs on the Linux operating system, because Linux is stable, performant, and less vulnerable to security breaches in comparison to other operating systems ('Windows 10')
5. YEROTH–ERP–3.0 has an user interface "Sales" to view sale information (Figure 2), and thus enables users to make managerial decisions
6. YEROTH–ERP–3.0 has an interface "Business dashboard" that generates financial accounting reports, from sale and payment information, to help managers to make "business decisions".

Sale date	Client name	Client location	Product name	Qty	Tax	Total sale
03.04.2022	DR.-ING. YEROTH YEROTH	YEROTH	test_yeroth_1	8.00	0.00	1,608.00
03.04.2022	DR.-ING. YEROTH YEROTH	YEROTH	test_yeroth_2	9.00	0.00	18,000.00
03.04.2022	PROF. DR.-ING. YEROTH	ANCIEN	test_yeroth_1	3.00	0.00	6,000.00
03.04.2022	DIVERS		test_yeroth_1	1.00	0.00	201.00
05.04.2022	DIVERS		test_yeroth_1	1.00	0.00	201.00
09.04.2022	DIVERS		test_yeroth_1	26.00	0.00	5,326.00
12.04.2022	DR.-ING. YEROTH YEROTH		MT_3334_CT_1	1.00	0.00	700.00
03.05.2022	DR.-ING. YEROTH YEROTH	YEROTH	YEROTH_TEST_CHAUSURE	2.00	37,700.00	557,700.00

Figure 2: Sale–information window.

5 Alert System

Users with roles « Administrator » or « Business manager » are the ones able to create alerts.

YEROTH–ERP–3.0 allows its users to create two types of alerts:

1. alerts over stocks–quantities
2. alerts over time intervals (this helps for perissable articles and for sales discounts over a period of time).

5.1 Alerts over Stock–Quantity

An alert over a stock–quantity is a message that is sent to a pre–determined user whenever "pre–determined" stock–quantity (X) of a specific article–stock is reached.

For instance, Xavier (« Business manager ») could create an alert for stock "mango" that will be triggered whenever stock "mango" quantity reaches 100; An alert–message is sent to user John (« Storekeeper »).

5.2 Alerts over Time–Period

A time–period is defined by a starting–date and an ending–date (dates are from the "gregorian" calendar).

An alert over a time–period (T) is a message that is generated, sent to a pre–determined user, and kept within YEROTH–ERP–3.0 from T's starting–date up to T's ending–date.

For example, an alert with a message has to be sent to Paul (« Cashier ») when the date of May 05th is reached. The alert message specifies that a rebate of 20% has to be applied on every sale of yoghourt 'trèsbon' during a time interval of 2 weeks.

6 Database Management System

YEROTH–ERP–3.0 uses 'MariaDB' as the standard DBMS. 'MariaDB' is very stable, very performant, and free–software.

7 Conclusion

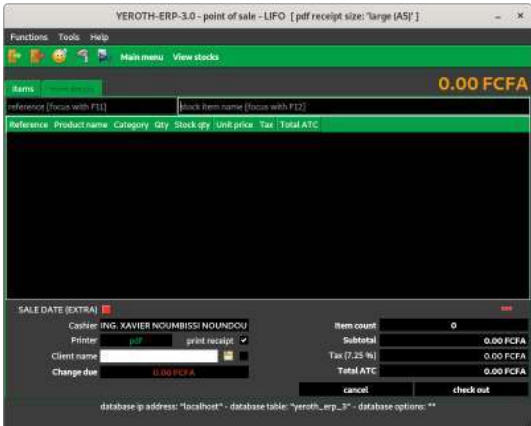


Figure 3: Point–of–sale window.

Figure 3 illustrates the window for selling articles.

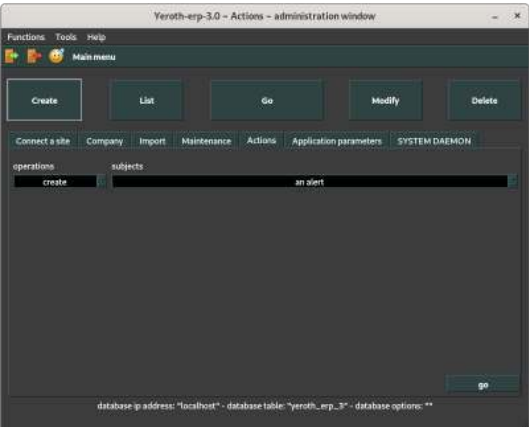


Figure 4: Administrative window for business manager.

Figure 4 illustrates the administrative window for business managers.

YEROTH-ERP-PGI-3.0 : Configuration MULTI-SITES (SUCCURSALES)

PROF. DR.-ING. DIPL.-INF. XAVIER NOUNDOU

CE LIVRET EXPLIQUE COMMENT RÉ-UTILISER YEROTH-PGI-ERP-3.0
AVEC DES SITES (SUCCURSALES).

VERSION : 6 novembre 2023

Table des matières

Table des matières	3
Table des figures	5
Liste des tableaux	7
1 INTRODUCTION	9
1.1 Définitions	9
1.1.1 Filiale (DICTIONNAIRE ROBERT)	9
1.1.2 Succursale (DICTIONNAIRE ROBERT)	9
2 Configuration D'1 ORDINATEUR D'1 SUCCURSALE	11
2.1 INSTALLATIONS DE yeroth-erp-pgi-3.0 PAR SUCCURSALE	12
3 CAS D'1 Filiale	13
3.1 TPE et PME	13
3.2 Très Grandes Entreprises (TGE)	14
4 CAS D'1 Succursale	15
4.1 Coupure de connexion Internet	16
5 LOGIN et connexion à 1 succursale	17
6 Bibliographie	19

Table des figures

2.1	FIGURE ILLUSTRATIVE de la marque succursale	11
3.1	BASE DE DONNÉES CENTRALISÉE d'1 filiale POUR TPE / PME	13
3.2	BASE DE DONNÉES CENTRALISÉE d'1 filiale pour TGE	14
4.1	BASE DE DONNÉES décentralisée du SIÈGE CENTRAL	15
5.1	Connexion à 1 succursale (SOCIÉTÉ – SITE)	17

Liste des tableaux

Chapitre 1

INTRODUCTION

LE PROGICIEL DE GESTION INTÉGRÉ YEROTH-PGI-3.0 [NOU22, NOU23a, NOU23b] permet à ses utilisateurs la réutilisation de la fonctionnalité "multi-sites (succursales)". La fonctionnalité "multi-sites (succursales)" permet à 1 organisation de contrôler ses activités décentralisées de façon centralisée à l'aide de YEROTH-PGI-3.0. Les opérations des succursales (ou encore d'1 filiale) sont répertoriées dans la base de données (MySQL) à l'aide de la colonne 'localisation'. 'La colonne localisation' de la base de données "yeroth_erp_3" **CORRESPOND À** :

1. Localisation (INSTALLATION EN FRANÇAIS)
2. Site (EN ANGLAIS)

dans les interfaces graphiques (GUI) de YEROTH-PGI-ERP-3.0.

1.1 Définitions

1.1.1 Filiale (DICTIONNAIRE ROBERT)

Société jouissant d'une personnalité juridique (à la différence de la succursale) mais dirigée ou contrôlée par une société mère.

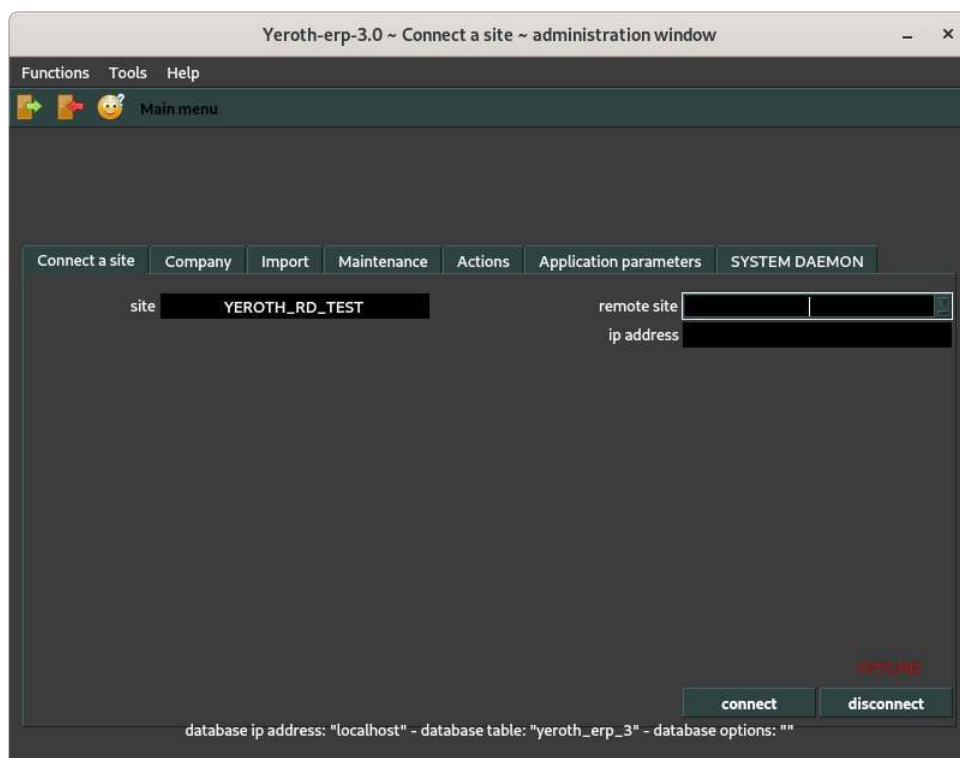
1.1.2 Succursale (DICTIONNAIRE ROBERT)

Établissement qui dépend d'un siège central, tout en jouissant d'une certaine autonomie. Exemple : Les succursales d'une banque.

Chapitre 2

Configuration D'1 ORDINATEUR D'1 SUCCURSALE

Figure 2.1 – FIGURE ILLUSTRATIVE de la marque succursale



CHAQUE ORDINATEUR INSTALLÉ DANS 1 SUCCURSALE possède 1 identification égale à 1 **CHAÎNE DE CARACTÈRE** unique.

CETTE CHAÎNE de caractère est la même pour chaque ordinateur de la succursale observée. LA FIGURE 2.1 illustre **L'IDENTIFIANT "YEROTH_RD_TEST"** d'1 succursale de test. Il s'agit de L'ONGLET "Connecter une localisation" dans la fenêtre 'FENÊTRE DE L'ADMINISTRATEUR'!

2.1 INSTALLATIONS DE yeroth-erp-pgi-3.0 PAR SUCCURSALE

1. Chaque copie de YEROTH-PGI-ERP-3.0 installée sur 1 ordinateur a été compilée avec l'identifiant de sa succursale.

L'IDENTIFIANT de la succursale se fait grâce à 1 chaîne de caractère dans le fichier nommé

`YEROTH_ERP_3_0_CURRENT_LOCALISATION_FOR_RELEASE_BUILD`

CE FICHIER EST À CE MOMENT DANS LE RÉPERTOIRE de développement :

`yeroth-erp-3-0/yeroth-erp-3-0-development-scripts`

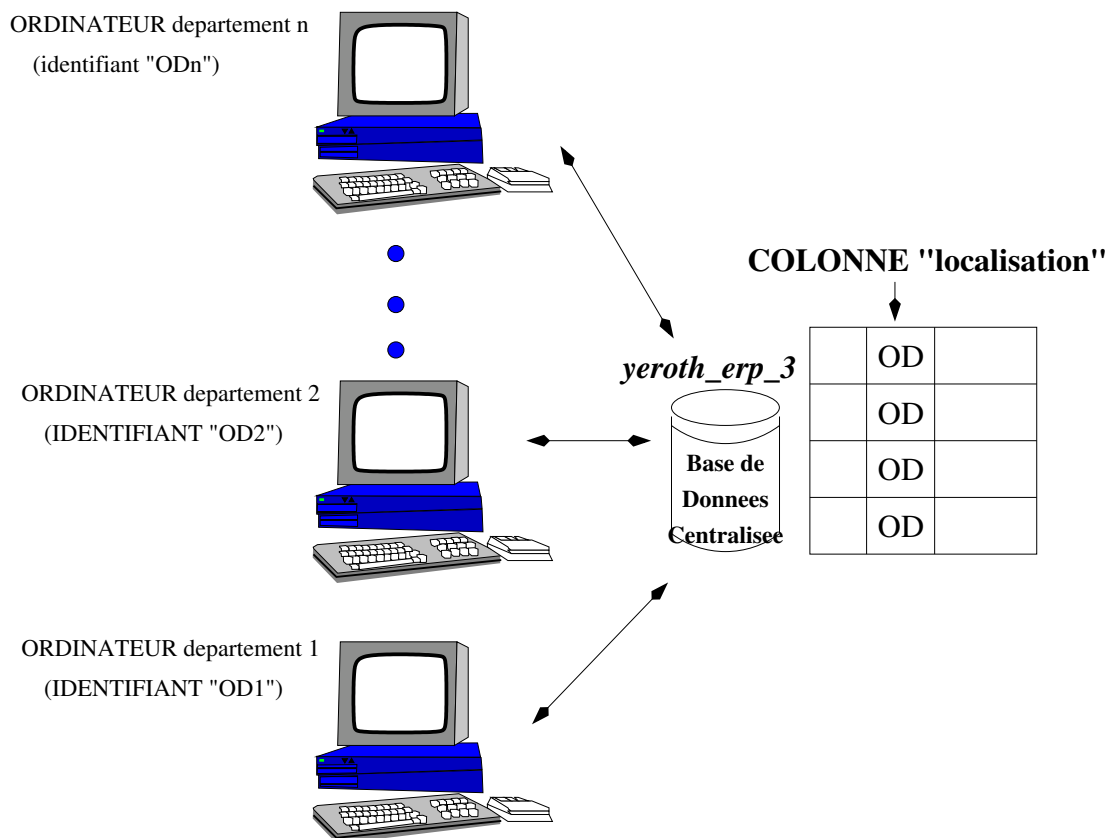
2. AINSI COMPILÉE, 1 copie exécutable de YEROTH-PGI-ERP-3.0 IDENTIFIE chacune de ses transactions dans la colonne **"localisation"** de la base de donnée MySQL d'1 IDENTIFIANT ÉGAL À TOUS LES IDENTIFIANTS de sa succursale.

Chapitre 3

CAS D'1 Filiale

3.1 TPE et PME

Figure 3.1 – BASE DE DONNÉES CENTRALISÉE d'1 filiale POUR TPE / PME

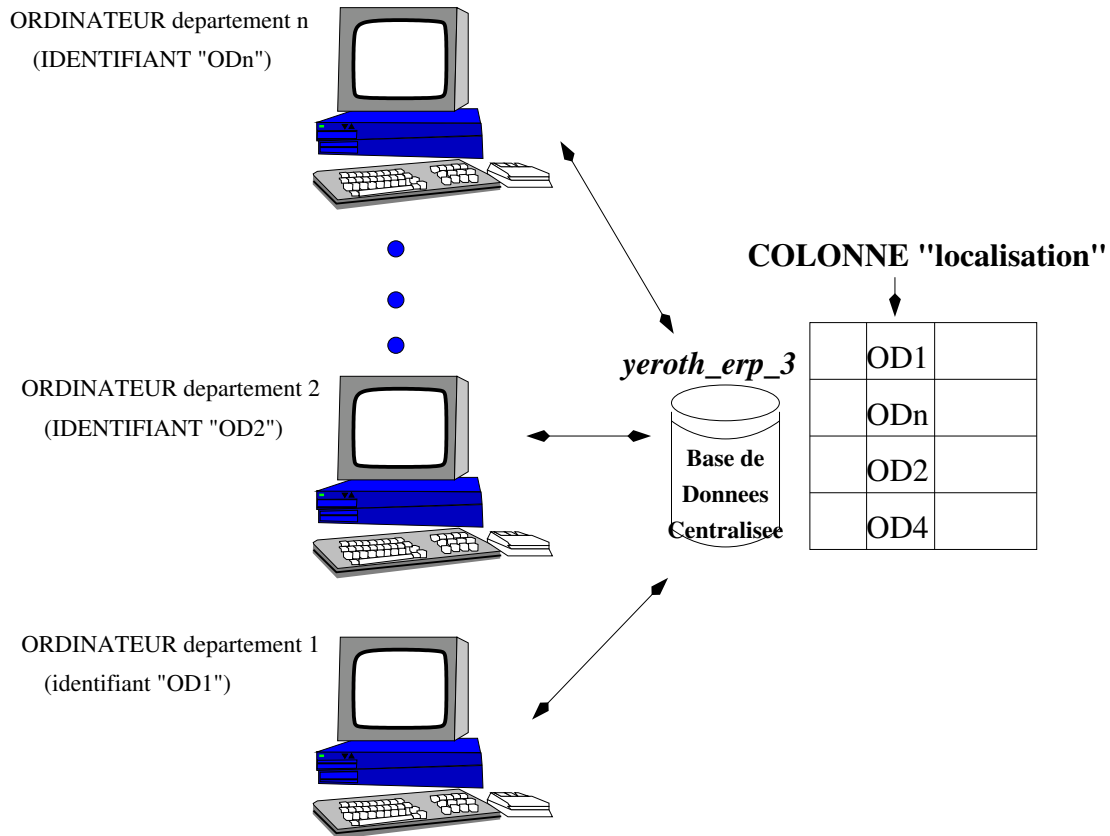


La figure 3.1 correspond à l'accès à la base de données de **LA Filiale (TPE ou PME)**.

DES TPE (Très Petite Entreprise) et des PME (Petite et Moyenne Entreprise) devraient suivre ce modèle avec 1 seul identifiant (pour colonne localisation) pour tous leurs ordinateurs ($OD1 = OD2 = \dots = ODn = OD$).

3.2 Très Grandes Entreprises (TGE)

Figure 3.2 – BASE DE DONNÉES CENTRALISÉE d'1 filiale pour TGE



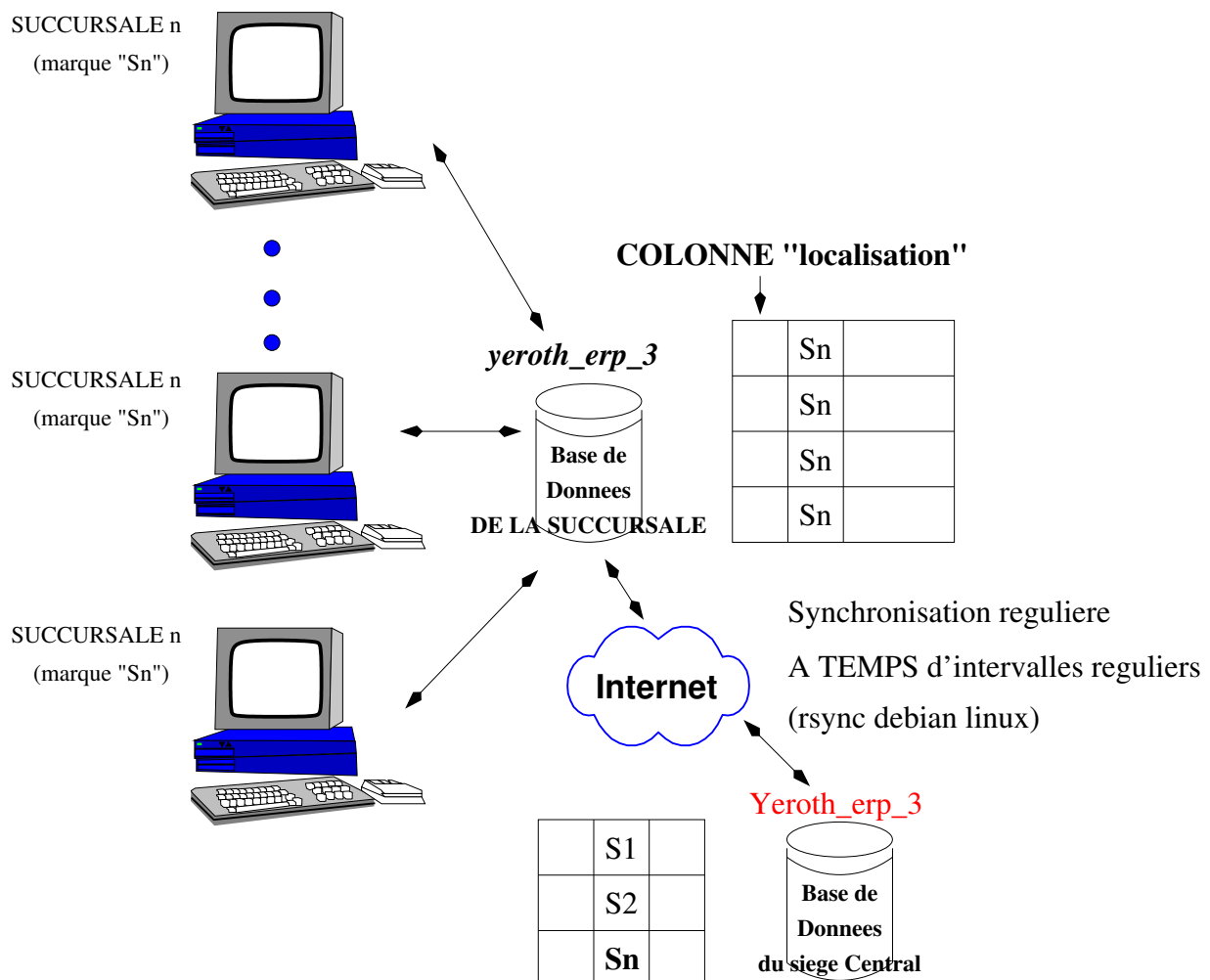
La figure 3.2 correspond à l'accès à la base de données de **LA Filiale TGE**.

DANS DE TRÈS GRANDES ENTREPRISES (Ex. : santa lucia AHALA, CAMWATER BONANJO, etc.), chaque département de L'ENTREPRISE A SON identifiant (pour colonne localisation) pour tous les ordinateurs du département ($OD1 \neq OD2 \neq \dots \neq ODn$).

Chapitre 4

CAS D'1 Succursale

Figure 4.1 – BASE DE DONNÉES décentralisée du SIÈGE CENTRAL



La figure 4.1 correspond à l'accès temps réel et temps différé aux bases de données RESPECTIVEMENT locale et centrale de LA SUCCURSALE et de la MAISON MÈRE (synchronisés à temps d'intervalles réguliers avec l'outil "RSYNC" DE [Debian Linux](#)).

4.1 Coupure de connexion Internet

L'ARCHITECTURE RÉSEAU PRÉSENTÉE dans la FIGURE 4.1 permet à 1 succursale de travailler localement, même en cas de coupure de connexion internet vers le Siège central.

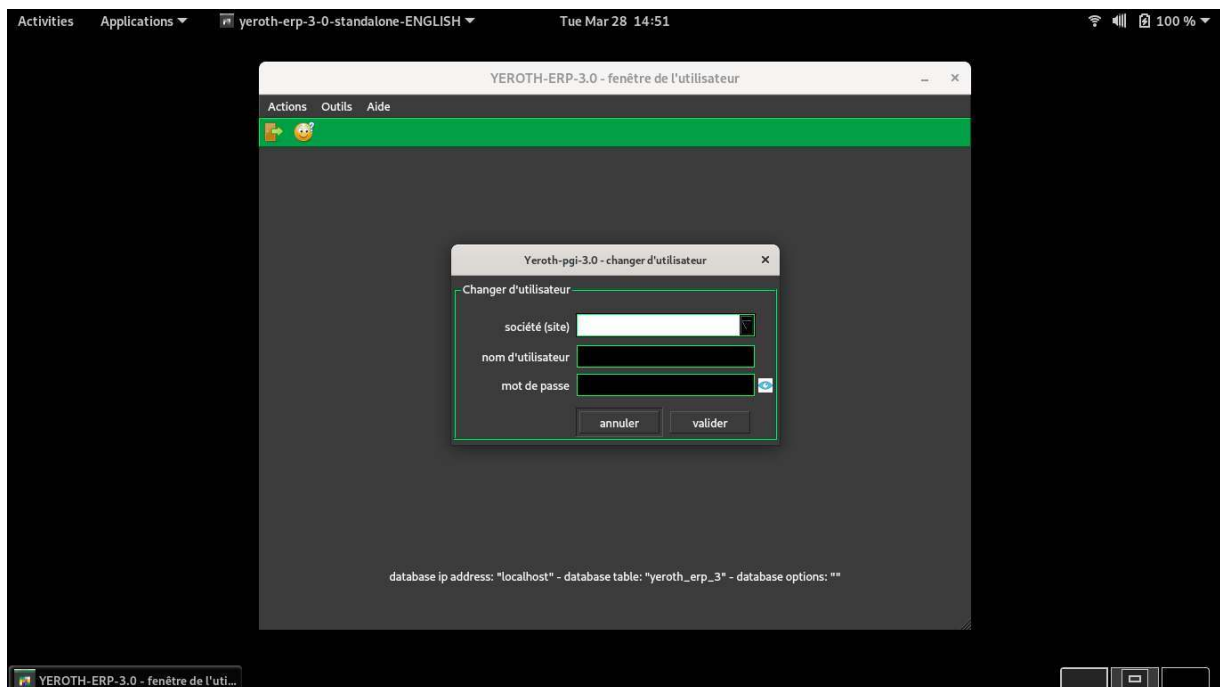
La synchronisation ultérieure permettrait ainsi de synchroniser des données avec le siège central via l'application **RSYNC-DEBIAN-LINUX**.

Chapitre 5

LOGIN et connexion à 1 succursale

Seul 1 utilisateur de type (ayant 1 rôle de) "*Manager*" ("*business manager*" en Anglais) a la possibilité de se connecter à 1 succursale différente de celle où son ordinateur est situé.

Figure 5.1 – Connexion à 1 succursale (SOCIÉTÉ – SITE)



La figure 5.1 illustre la fenêtre dialogue de connexion de YEROTH-PGI-ERP-3.0 pour 1 utilisateur. Cette fenêtre contient 1 champs de texte duquel 1 utilisateur ayant 1 rôle de *Manager* peut choisir 1 succursale vers laquelle il peut se connecter.

Lorsqu'un utilisateur est connecté à 1 autre succursale, toutes des informations et données affichées à l'écran proviennent de la base de données de la succursale concernée.

Chapitre 6

Bibliographie

- [NOU22] XAVIER NOUNDOU. YEROTH-ERP-PGI-3.0 DOCTORAT COMPENDIUM. https://archive.org/download/yeroth-erp-pgi-compendium_202206/JH_NISSI_ERP_PGI_COMPENDIUM.pdf, 2022. ACCÉDER POUR LA DERNIÈRE FOIS le 29 Mars 2023 à 09:00.
- [NOU23a] XAVIER NOUNDOU. GUIDE D'INSTALLATION POUR LE PROGICIEL DE GESTION INTÉGRÉ YEROTH-PGI-3.0. https://archive.org/download/yeroth-erp-3-0-guide-dinstallation-standalone_202303/yeroth-erp-3-0-guide-dinstallation-standalone.pdf, 2023. ACCÉDER POUR LA DERNIÈRE FOIS le 29 mars 2023 à 12:00.
- [NOU23b] XAVIER NOUNDOU. INSTALLATION GUIDE FOR ERP SOFTWARE SYSTEM YEROTH-ERP-3.0. https://archive.org/download/yeroth-erp-3-0-installation-guide-standalone_202304/yeroth-erp-3-0-installation-guide-standalone.pdf, 2023. ACCESSED LAST TIME ON APRIL 18, 2023 at 08:05.

Runtime Verification Of SQL Correctness Properties with **YR_DB_RUNTIME_VERIF**

Xavier N. Noundou¹

¹Yaoundé, Center, Cameroon.

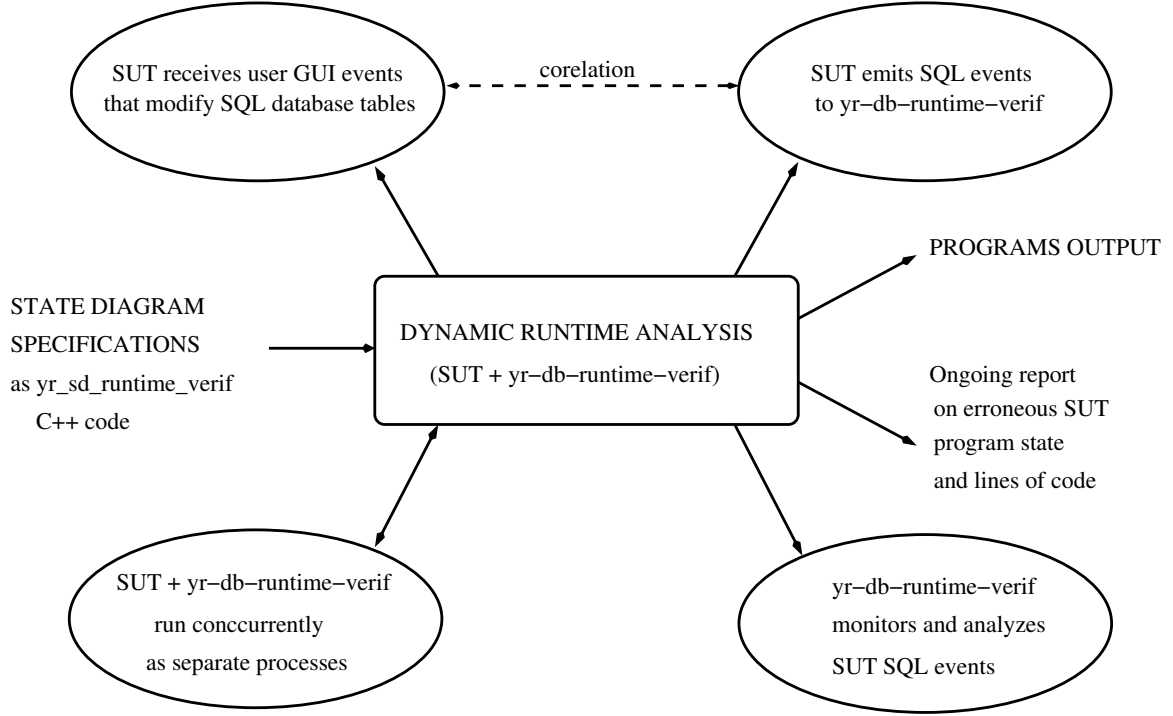
Contributing authors: yeroth.d@gmail.com;

Abstract

Software correctness properties are essential to maintain quality by continuous and regressive integration testing, as well as runtime monitoring the program after customer deployment. This paper presents an effective and lightweight C++ program verification framework: **YR_DB_RUNTIME_VERIF**, to check SQL (Structure Query Language) [1] software correctness properties specified as temporal safety properties [2]. A temporal safety property specifies what behavior shall not occur, in a software, as sequence of program events. **YR_DB_RUNTIME_VERIF** allows specification of a SQL temporal safety property by means of a state diagram mealy machine [3]. In **YR_DB_RUNTIME_VERIF**, a specification characterizes effects of program events (via SQL statements) on database table columns by means of set interface operations (\in , \notin), and, enable to check these characteristics hold or not at runtime. Integration testing is achieved for instance by expressing a state diagram that encompasses both Graphical User Interface (GUI) states and MySQL [4] databases queries that glue them. For example, a simple specification would encompass states between 'Department administration' and 'Stock listing' GUI interfaces, and transitions between them by means of MySQL databases operations. **YR_DB_RUNTIME_VERIF** doesn't generate false warnings; **YR_DB_RUNTIME_VERIF** specifications are *not desirable (forbidden) specifications (fail traces)*. This paper focuses its examples on MySQL database specifications, labeled as states diagrams events, for the newly developed and FOSS (Free and Open Source Software) Enterprise Resource Planing Software YEROTH-ERP-3.0 [5].

Keywords: model-based testing, reactive system analysis, computer software program analysis, computer software dynamic program analysis, software integration testing with SQL and GUI, runtime monitoring

Fig. 1: YR_DB_RUNTIME_VERIF WORKFLOW (diagram inspired from operation diagram in [6]).



1 Introduction

Table 1: YEROTH-ERP-3.0 RELEVANT SOFTWARE SYSTEM METRICS

Software System Metric	Value
User Interface (windows, dialog) number	60
MariaDB SQL table number	38
MariaDB SQL table column number	320
Source lines of code (SLOC)	300,000

1.1 Motivations

This paper describes an effective dynamic analysis framework, based on runtime monitors specified in C++ programs (implemented in the software library `yr_sd_runtime_verif`), to perform software temporal safety property checking of GUI (Graphical User Interface) based software.

GUI based software are very comfortable and handy to use. However, tools to perform temporal safety property verification of GUI software

are almost not available as FOSS. The testing of combinations between GUI windows and database queries that glue them to make sense to the user, is almost unavailable as FOSS, or at all to the best of the knowledge of the author of this paper. The FOSS C++ library `libfsmtest` [7] provides test suite generation support for source code behavior specifications as mealy automata. However, `libfsmtest` only allows for *desirable* correctness properties, and doesn't provide GUI (interaction) support or as plugin-based.

Unit or integration testing for GUI widgets is available by use of "NUnit" testing frameworks like e.g. `Qt-Test` [8], `CppUnit` [9], etc.. Software testing across GUI widgets (and MySQL queries) is however limited in support by these "NUnit" framework. To the best of the knowledge of the author of this paper, `DejaVu` [10] provides some support for `Java`'record and replay' testing while `FROGLOGIC` [11] provides support for C++ GUI software 'record and replay' testing technology. 'Record and replay' testing means a user performs a sequence of events that are recorded by testing infrastructure and automatically replay later

on to see if expected events thereof occur. However, none of this 'record and replay' technology tool enable temporal safety property specification as FOSS, with SQL as plugin.

As we will see in the related work, section 7, of this paper, most of software correctness property checking frameworks don't put an emphasis on checking temporal safety property of GUI software. Characterizing the effects of program statements (via SQL statements) on database table columns, and to check that these characteristics hold or not, is of predominant importance for large software systems with an impressive number of database tables. Table 1 illustrates for instance FOSS YEROTH-ERP-3.0 relevant software system metrics.

It means it can be very difficult for developers to keep application related logical requirements between the tables without appropriate software testing or analysis tools.

A large amount of former work on runtime monitoring assumes for a sequential program, or an abstraction of the program as one single source code, on which program analysis is performed [12–16].

The program analysis technique the author of this paper presents here abstract SQL events, GUI events, or sequences of them, as a state diagram, and enables developers to run them sequentially against a runtime monitor specified as a C++ program. In particular, the example presented in Section 2 specifies results of GUI windows events as SQL database pre-conditions on state diagram transitions; SQL events are specified as state diagram transition events. Figure 1 shows a high level overview of **YR_DB_RUNTIME_VERIF** workflow.

1.2 Main Contributions

This paper presents 3 original main contributions:

- an industrial level quality framework (**YR_DB_RUNTIME_VERIF**: <https://github.com/yerothd/yr-db-runtime-verif>), that solves temporal property verification by dynamic program analysis. **YR_DB_RUNTIME_VERIF** makes use of the C++ Qt-DBus library, to input a *runtime monitor specification* (**yr_sd_runtime_verif**) as C++ program code, that also enables software-library-plugin checks;

- a C++ library: **yr_sd_runtime_verif** (https://github.com/yerothd/yr_sd_runtime_verif); modeling a state diagram runtime monitoring interface using only set algebra inclusion operations (\in , \notin) for state diagram program state specification as pre- and post-conditions.

yr_sd_runtime_verif only enables the specification of states diagrams specifications as *not desirable (forbidden) behavior specifications (fail traces)*. Thus, **YR_DB_RUNTIME_VERIF** doesn't generate any false warning. A violation of a safety rule has been found whenever a final state could be reached. On the other hand, not reaching a final state doesn't mean that there is not a test case (or test input) that cannot reach this final state.

- An application of **YR_DB_RUNTIME_VERIF** to check 1 temporal safety property error, found in the ERP FOSS YEROTH-ERP-3.0.

Previous version of this paper

This paper extends a previous version [17], currently in conference proceedings **SPLASH-ICTSS 2023** submission, with state diagram with more than 2 states, guarded conditions specifications, 2 new keywords for state diagram transition trace specification ("**in_sql_event_log**", "**not_in_sql_event_log**"), and **YR_DB_RUNTIME_VERIF** binaries with more than 1 runtime monitor.

1.3 Overview

This paper is organized as follows: Section 2 presents a motivating example that will be used throughout this paper to explain the presented concepts of this paper. Section 3 presents formal definitions of the principal concepts used in this paper. Section 4 presents the software architecture of **YR_DB_RUNTIME_VERIF**, our GUI dynamic analysis framework. Section 5 introduces the C++ software library **yr_sd_runtime_verif** to model states diagrams, and reused by **YR_DB_RUNTIME_VERIF**. We evaluate our dynamic runtime analysis in Section 6. Section 7 compares this paper with other papers that achieve similar work or endeavors. Section 8 concludes this paper.

Fig. 2: A motivating example, as current bug in YEROTH-ERP-3.0.

$\overline{Q0} := \text{NOT_IN_PRE}(\text{YR_ASSET}, \text{department.department_name}).$
 $\overline{Q1} := \text{IN_POST}(\text{YR_ASSET}, \text{stocks.department_name}).$



Fig. 3: YEROTH-ERP-3.0 administration section displaying departments ($\neg Q0$).

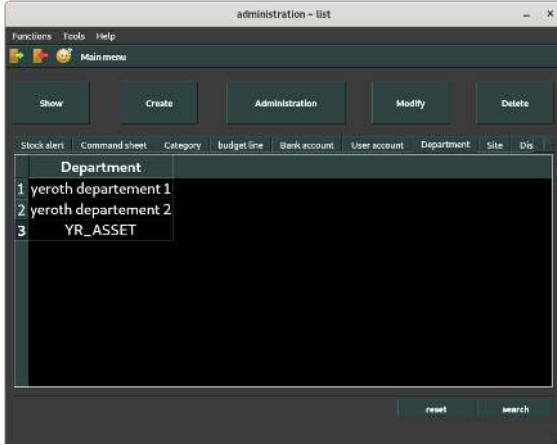
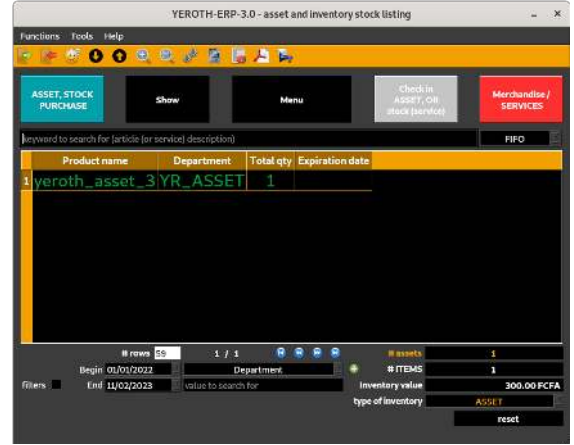


Fig. 4: YEROTH-ERP-3.0 stock asset window listing some assets ($\overline{Q1}$).



2 Motivating Example: missing department definition

2.1 The Enterprise Resource Planing Software YEROTH-ERP-3.0

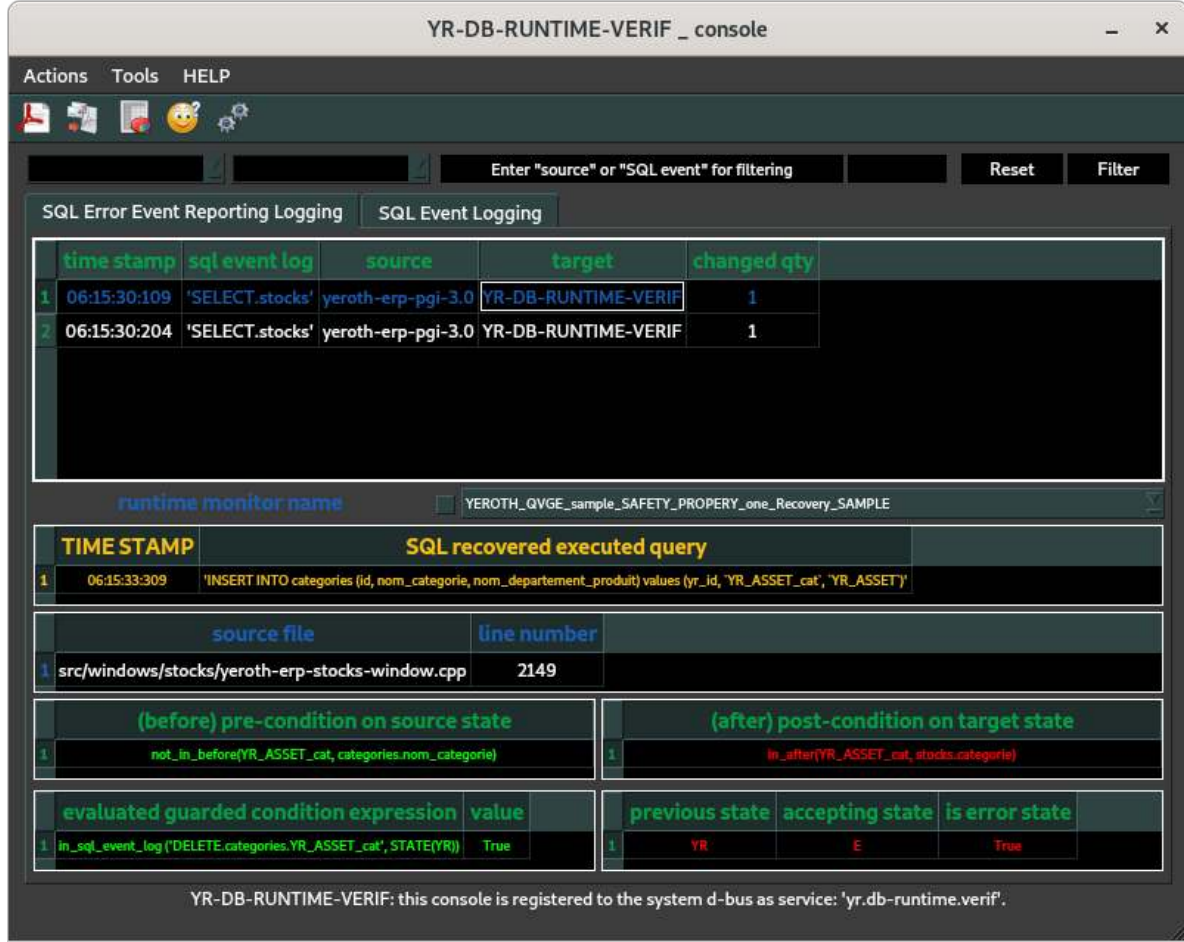
YEROTH-ERP-3.0 is a fast, yet very simple in terms of usage, installation, and configuration Enterprise Resource Planing Software developed by Noundou et al. [5] for very small, small, medium, and large enterprises. YEROTH-ERP-3.0 is developed using C++ by means of the Qt development library. YEROTH-ERP-3.0 is a large software with around 300 000 (three hundred thousands) of physical source lines of code. **YR_DB_RUNTIME_VERIFY** could be used for integration

testing of YEROTH-ERP-3.0, among different software modules.

2.2 Example Temporal Safety Property

The motivating example of this paper consists of the temporal safety property stipulating that **"A DEPARTMENT SHALL NOT BE DELETED WHENEVER STOCKS ASSET STILL EXISTS UNDER THIS DEPARTMENT"**. This statement means that a user shall be denied the removal of department 'YR_ASSET' in Figure 3 because there are still a stock asset listed within department 'YR_ASSET', as illustrated in Figure 4. Figure 2

Fig. 5: `YR_DB_RUNTIME_VERIF` command line shell output demonstrating that a final state has been reached (Section 6 analyzes these results).



illustrates the above temporal safety property as a simple state diagram.

2.2.1 State Diagram Explanation

'D' is a *start* state as illustrated by an arrow ending on its state shape. 'E' is a *final* (error, or accepting) state as illustrated by a double circle as state shape.

The pre-condition Q_0 (as a predicate) in state 'D':

"NOT_IN_PRE(YR_ASSET, departement.department_name)" means:

- a department named 'YR_ASSET' is not in column 'department_name' of MariaDB SQL database table 'department'. This might happen whenever

button 'Delete' in Figure 3 is pressed when item 'YR_ASSET' is selected.

Similarly, the post-condition $\overline{Q_1}$ (as a predicate) "IN_POST(YR_ASSET, stocks.department_name)", in accepting state 'E', means:

- a department named 'YR_ASSET' is in column "department_name" of MariaDB SQL database table 'stocks'.

The state diagram event transition in Figure 2: 'SELECT.department' denotes that when in 'D', a SQL 'select' on database table "department" has occurred; 'E' is then reached as an *accepting state*.

Guarded Condition Expression

The guarded condition expression "[`in_set_trace ('DELETE.department.YR_ASSET', STATE(D))`]" means a SQL 'DELETE' event removing a department named 'YR_ASSET' from MariaDB SQL table 'department' must have occurred in the trace leading to state 'D'.

Yr_sd_runtime_verif Specification Code

The source code specified in Listing 2 also illustrates a specification in C++ using software library `yr_sd_runtime_verif` of the state diagram specification above.

2.3 YR_DB_RUNTIME_VERIF Analysis Report

The motivating example automaton in Figure 2 is analyzed by **YR_DB_RUNTIME_VERIF** as follows:

- whenever department 'YR_ASSET' is deleted in YEROTH-ERP-3.0, as done in Figure 3, the runtime monitor state 'D' with a state condition $Q0$ is entered
- when MySQL library (plugin) event 'SELECT.department' occurs, in Figure 3 because of YEROTH-ERP-3.0 displaying the remaining product departments, the guarded condition for edge event 'SELECT.department' is automatically evaluated to 'True' by C++ library `yr_sd_runtime_verif`, because no other guarded condition was specified by the developer
- `yr_sd_runtime_verif` enters the runtime monitor state to 'E' and state condition $\overline{Q1}$ via method `YR_trigger_an_edge_event(QString an_edge_event)` because there are still assets (`yeroth_asset_3`) left within product department 'YR_ASSET', as illustrated in Figure 4. 'E' is then an accepting (or final or error) state.

Figure 5 illustrates an analysis result of the afore described process, which gets evaluated and described in Evaluation Section 6.

3 Formal Definitions

`yr_sd_runtime_verif`'s formal description of the state diagram formalism follows *Mealy machine* [3] added with *accepting states (final or erroneous state), and state diagram transition pre- and post-conditions*. Another excellent, detailed with proofs and theory presentation of mealy automata [18] is available. In comparison to statechart [19], which is a *visual formalism* for states diagrams, `yr_sd_runtime_verif` doesn't support for instance the following features: *hierarchical states (composite state, submachine state), timing conditions*.

Definition 1.

A state diagram is a 8-tuple $(S, S_0, C, \Sigma, \Lambda, \delta, T, \Gamma)$ where:

- S : a finite set of states
- $S_0 \in S$: a start state (or initial state)
- C : a set of predicate conditions; pre-conditions are underlined (e.g.: $\underline{Q0}$), and post-conditions are overlined (e.g.: $\overline{Q1}$). A pre-condition is comparable to a Harel-statechart *guarded condition*.
- Σ : an input alphabet, $\Sigma := \{False, True\}$.
'False' means no input from SUT into **YR_DB_RUNTIME_VERIF**.
'True' means any input could come from SUT.
- Λ : an output alphabet (of program events $e_n (n \in \mathbb{N})$), ϕ the no program event. A program event generally corresponds to a function or method call at a SUT source code statement (or program point).
- $\delta : S \times C$: a 2-ary relation that maps a state s to a state-condition c as either a state diagram transition pre-condition (\underline{c}), or as a state diagram transition post-condition (\overline{c}).
- $T : S \times \Sigma \rightarrow S \times \Lambda$: a transition function that maps an input symbol to an output symbol and the next state.
- δ : a 2-ary relation that maps a state diagram transition to a guarded condition expression.
- Γ : a set of accepting states; $\Gamma \in S$.

For instance, for the motivating example described in Figure 2 we have:

- $S = \{D, E\}$;
- $S_0 = D$;
- $C = \{Q0, \overline{Q1}\}$;
- $\Sigma = \{False, True\}$;
- $\Lambda = \{\phi, 'SELECT.department'\}$;
- $\delta = \{(D, \underline{Q0}), (E, \overline{Q1})\}$;
- $T = \{((D, False), (D, \phi)), ((D, True), (E, 'SELECT.department'))\}$;
- $\Gamma = \{E\}$

Definition 2.

A pre-condition of a state diagram transition is a predicate that must be true before the transition can be triggered. A pre-condition $\underline{Q0}$ could have 2 forms:

- $\underline{Q0} := \text{IN_PRE}(X, Y)$ that means value "X" is in (\in) database column value set "Y".
- $\underline{Q0} := \text{NOT_IN_PRE}(X, Y)$ that means value "X" is not in (\notin) database column value set "Y".

Definition 3.

A post-condition of a state diagram transition is a predicate that must be true after the transition was triggered. A post-condition $\overline{Q1}$ could have 2 forms:

- $\overline{Q1} := \text{IN_POST}(A, B)$ that means value "A" is in (\in) database column value set "B".
- $\overline{Q1} := \text{NOT_IN_POST}(A, B)$ that means value "A" is not in (\notin) database column value set "B".

For state diagram mealy machines with more than 2 states, only the first transition has a pre-condition specification (**IN_PRE**, or **NOT_IN_PRE**). Each other transition only has a post-condition specification (**IN_POST**, or **NOT_IN_POST**). Since each state only has 1 outgoing (edge) state transition, the post-condition of the previous (incoming) state transition acts as the pre-condition of the next transition.

Definition 4.

A trace $T_n = \langle e^0, e^1, \dots, e^n \rangle$ is a sequence of SUT events (or SUT program points) $e^i, i \in \{0, \dots, n\}$ of length n . $\text{trace}(D)$ is the trace of SUT events up to state D. For instance, for the motivating example described in Figure 2 we have: $\text{trace}(E) = \text{trace}(D), \langle 'SELECT.department' \rangle$.

Proposition 1: NO FALSE WARNINGS.

`yr_sd_runtime_verif` only allows 1 outgoing edge or transition for a state in its specifications, and for *not desirable (forbidden) behavior*, as illustrated in Figure 2. There is no need to specify the red colored edge in Figure 2 because it represents runtime cases where no input events arrive from SUT into **YR_DB_RUNTIME_VERIF**. These 2 properties, together with algorithm `'YR_trigger_an_edge_event(QString an_edge_event)'` (Listing 3) of `yr_sd_runtime_verif`, ensures that there are no false warnings during **YR_DB_RUNTIME_VERIF** analyses. For example, the runtime monitoring or verification systems [12–16] may give false warnings.

3.1 Guarded Condition Expression Specification in `yr_sd_runtime_verif`

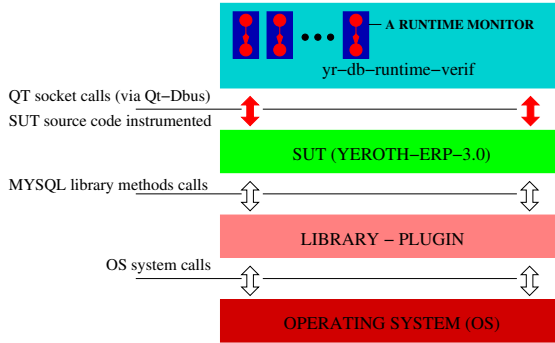
Guarded conditions expressions can be specified using one of the `yr_create_monitor_edge` method and a boolean expression of type **YR_CPP_BOOLEAN_expression**. An edge without an explicit guarded condition has an *implicit* `'[True]'` guarded condition on it. The implicit guarded condition `'[True]'` mustn't be identified as an implicit input event `'True'`, as specified in Definition 1.

Guarded conditions are meant to be trace set specification on program events. For instance in Figure 2 (motivating example): `"[in_set_trace ('DELETE.department.YR_ASSET', STATE(D))]"` means that a SQL `'DELETE'` event removing a department named `'YR_ASSET'` from MariaDB SQL table `'department'` must have occurred in the trace leading to state `'D'`, before event `'SELECT.department'` can be triggered. A guarded condition could have two practical forms:

- `"[in_set_trace ('event', STATE(D))]"` is equivalent to: `'event' ∈ trace(D)`.
- `"[not_in_set_trace ('event', STATE(D))]"` is equivalent to: `'event' ∉ trace(D)`.

where `'event'` is an input event ($\text{event} \in \Sigma$) and `'D'` a state diagram state ($D \in \mathbf{S}$).

Fig. 6: **YR_DB_RUNTIME_VERIF**: simplified software system architecture.



4 The Software Architecture of **YR-DB-RUNTIME-VERIF**

4.1 Dynamic Analysis

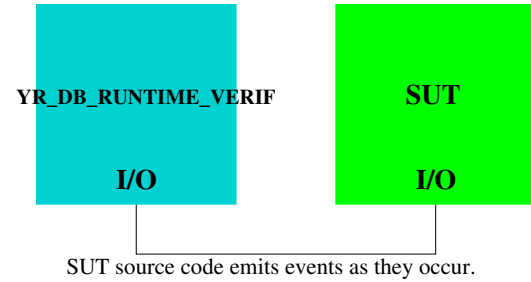
4.1.1 SUT Source Code Instrumentation.

YR_DB_RUNTIME_VERIF runs as a separate Debian Linux process from the application to dynamically analyze (YEROTH-ERP-3.0 in this case). Figure 6 illustrates a software system architecture layer of a software system that uses **YR_DB_RUNTIME_VERIF**. Figure 6 and Figure 7 illustrate how YEROTH-ERP-3.0 is instrumented to send MySQL database events, as they occur on due to the GUI of YEROTH-ERP-3.0, to process **YR_DB_RUNTIME_VERIF**, so it can perform runtime analysis of the monitor implemented within it.

4.1.2 Debugging Information.

Each GUI manipulation of YEROTH-ERP-3.0 in its instrumented source code part could generate a state transition within the analyzed runtime monitor state diagram in **YR_DB_RUNTIME_VERIF**. Visualize "line 35" of Figure 5 to observe that a specific analysis message is sent to the console of **YR_DB_RUNTIME_VERIF** in cases where a final state has been reached; the message at "line 33" is for an accepting (final) state of the state diagram specification of the motivating example presented in Figure 2.

Fig. 7: **YR_DB_RUNTIME_VERIF** and SUT socket communication (diagram inspired from Jan Peleska diagram-work).



4.2 SQL Events

YR_DB_RUNTIME_VERIF currently only analyzes the 4 SQL events in Table 2.

4.3 A Runtime Monitor (An Analysis Client)

Listing 1: "XML file adaptor for YEROTH-ERP-3.0 test cases (reduced from 4 to only 1 SQL event for paper)."

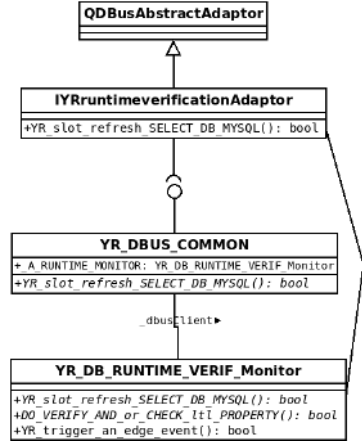
```
<!DOCTYPE node PUBLIC "-//freedesktop//
DTD D-BUS Object Introspection 1.0//EN"
"http://www.freedesktop.org/standards/
dbus/1.0/introspect.dtd">
<node name="/YRruntimeverification">
  <interface name="com.yeroth.rd.
  IYRruntimeverification">
    <method name="
      YR_slot_refresh_SELECT_DB_MYSQL
">
      <annotation name="org.qtproject.QtDBus.
      QtTypeName.In0" value="QString"/>
      <annotation name="org.qtproject.QtDBus.
      QtTypeName.In1" value="uint"/>
      <annotation name="org.qtproject.QtDBus.
      QtTypeName.In2" value="bool"/>
      <arg type="QString" direction="in"/>
      <arg type="uint" direction="in"/>
      <arg type="bool" direction="out"/>
    </method>
  </interface>
</node>
```

An user (an analysis client) of **YR_DB_RUNTIME_VERIF** needs to subclass class **YR_DB_RUNTIME_VERIF_Monitor**. The UML class diagram in Figure 8 displays the class structure

Table 2: SQL Event Dbus Method Interface

SQL Event	Dbus Method Interface
DELETE	YR_slot_refresh_DELETE_DB_MYSQL(QString, uint)
INSERT	YR_slot_refresh_INSERT_DB_MYSQL(QString, uint)
UPDATE	YR_slot_refresh_UPDATE_DB_MYSQL(QString, uint)
SELECT	YR_slot_refresh_SELECT_DB_MYSQL(QString, uint)

Fig. 8: **YR_DB_RUNTIME_VERIF**: simplified class diagram in UML [20].



of **YR_DB_RUNTIME_VERIF**. Qt-Dbus communication adaptor **IYRruntimeverificationAdaptor** shall be generated by the user of this library (on **YR_DB_RUNTIME_VERIF** side) using Qt-Dbus command **qdbusxml2cpp** and an XML file, similar to the one displayed in Listing 1:

An analysis client must first override method **'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY'** of class **'YR_DB_RUNTIME_VERIF_Monitor'** so to implement a checking algorithm for each event received from SUT, as for instance the events illustrated in Figure 2 of the motivating example. The analysis client then calls method **'YR_trigger_an_edge_event(QString an_edge_event)'** (Listing 3) of class **'YR_CPP_RUNTIME_MONITOR'** of C++ library **yr_sd_runtime_verif** for each corresponding state diagram transition event.

Fig. 9: Class diagram in UML [20] to model a State Transition Diagram.

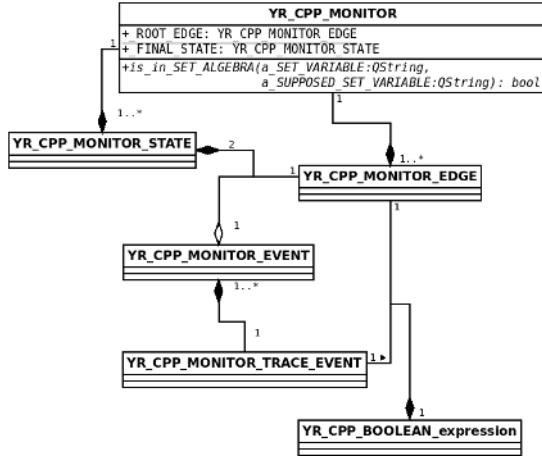
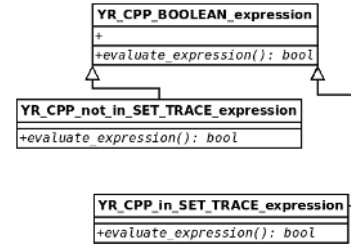


Fig. 10: Class diagram in UML [20] to model state diagram transition trace conditions in yr_sd_runtime_verif code.



Listing 2: yr_sd_runtime_verif C++ code modeling a current bug in YEROTH-ERP-3.0 (Figure 2).

```

1 YR_CPP_MONITOR_EDGE *a_last_edge_0 = create_yr_monitor_edge("D",
2 "E",
3 "select.departements_produits");
4
5 a_last_edge_0->get_SOURCE_STATE()->set_START_STATE(true);
6
7 a_last_edge_0->get_TARGET_STATE()->set_FINAL_STATE(true);
8
9 a_last_edge_0->set_PRE_CONDITION_notIN("YR_ASSET",
10 "departements_produits.nom_departement_produit");
11
12 a_last_edge_0->set_POST_CONDITION_IN("YR_ASSET",
13 "stocks.nom_departement_produit");
14
15 YR_register_set_final_state_CALLBACK_FUNCTION(&YR_CALL_BACK_final_state);

```

5 yr_sd_runtime_verif: A C++ Library to Model States Diagrams

5.1 Structure Of yr_sd_runtime_verif

yr_sd_runtime_verif is a state diagram C++ library the author of this paper created to work with the dynamic analysis program YR_DB_RUNTIME_VERIF. Figure 9 and Figure 10 represent the class structure, in UML, of yr_sd_runtime_verif. Listing 2 shows the C++ code that models the motivating example in

Figure 2, and that uses runtime monitoring C++ state diagram library yr_sd_runtime_verif.

There is no need to write C++ code for the red specified edge of Figure 2; this represents runtime cases where no input event arrives from SUT into YR_DB_RUNTIME_VERIF.

Table 3 specifies which class is in yr_sd_runtime_verif code for each runtime monitor/state diagram element.

5.2 Methods for Pre- and Post-Condition Specifications

Table 4 illustrates methods for specifying pre- and post-conditions of a runtime monitor

Table 3: Runtime Monitor Specification Classes

State Diagram Feature	Class
State	YR_CPP_MONITOR_STATE
Transition	YR_CPP_MONITOR_EDGE
Event	YR_CPP_MONITOR_EVENT
Trace at state level	YR_CPP_MONITOR_TRACE_EVENT
Guard Condition	YR_CPP_BOOLEAN_expression
Set Trace Inclusion at edges	YR_CPP_in_SET_TRACE_expression
Set Trace non Inclusion at edges	YR_CPP_not_in_SET_TRACE_expression
Runtime Monitor	YR_CPP_MONITOR

Table 4: yr_sd_runtime_verif Methods for Pre-/Post-Condition Specification

Class YR_CPP_MONITOR_EDGE Methods	Utility
set_PRE_CONDITION_notIN (QString, QString)	sets a NOT IN DATABASE pre-condition
set_PRE_CONDITION_IN (QString, QString)	sets an IN DATABASE pre-condition
set_POST_CONDITION_notIN (QString, QString)	sets a NOT IN DATABASE post-condition
set_POST_CONDITION_IN (QString, QString)	sets an IN DATABASE pre-condition

state diagram transition. Each method takes in 2 arguments of string ('QString') type: 'DB_VARIABLE', 'db_TABLE__db_COLUMN'.

The first method argument: 'DB_VARIABLE', specifies which variable is to be expected as value for the specification of the second variable argument 'db_TABLE__db_COLUMN'. The second variable gives in a string to be specified in format "DB_table_name.DB_table_column"; and its supposed value is the returned value of the first variable argument 'DB_VARIABLE'.

These 4 pre- and post-conditions methods make assumptions that a **program variable value** 'DB_VARIABLE' is in set "DB_table_name.DB_table_column" or not; if the value of 'DB_VARIABLE' is in the database table column, it means it is **in the set** (\in) of values "DB_table_name.DB_table_column"; and not being in the table column means it is **not in the set** (\notin).

Example from the motivating example in Section 2

Listing 2 of the runtime monitoring specification stipulates for instance in its "line 12", as post-condition:

```
a_last_edge_0->
    set_POST_CONDITION_IN("YR_ASSET",
        "stocks.nom_departement_produit");
that 'YR_ASSET' shall be a value in the
value set ( $\in$ ) of SQL table 'stocks' column
'nom_departement_produit'.
```

5.3 SUT Event Processing Method YR_trigger_an_edge_event

Listing 3 illustrates the pseudo-code of yr_sd_runtime_verif SUT event processing method YR_trigger_an_edge_event(QString an_edge_event). 'YR_trigger_an_edge_event(QString an_edge_event)' is responsible for interpreting a monitor at runtime, based on its current state, and on the current event received from SUT. Each state in yr_sd_runtime_verif states diagrams

Listing 3: C++ Pseudo-code for `YR_trigger_an_edge_event(QString an_edge_event):`
`yr_sd_runtime_verif` method for triggering state diagram events (edges or transitions).

```

1  bool MONITOR::YR_trigger_an_edge_event(QString an_edge_event)
2  {
3      MONITOR_EDGE cur_OUTGOING_EDGE = _cur_STATE.outgoing_edge();
4
5      if (cur_OUTGOING_EDGE.evaluate_GUARDED_CONDITION_expression() &&
6          (an_edge_event == cur_OUTGOING_EDGE.edge_event_token()))
7      {
8          bool precondition_IS_TRUE = cur_OUTGOING_EDGE
9              .CHECK_SOURCE_STATE_PRE_CONDITION(_cur_STATE);
10
11         if (precondition_IS_TRUE)
12         {
13             set_current_triggered_EDGE(cur_OUTGOING_EDGE);
14
15             MONITOR_STATE a_potential_accepting_state =
16                 cur_OUTGOING_EDGE.get_TARGET_STATE();
17
18             if (CHECK_whether_STATE_is_Final(a_potential_accepting_state))
19             {
20                 CALL_BACK_final_state_FUNCTION(a_potential_accepting_state);
21             }
22             return true;
23         }
24     }
25     return false;
26 }

```

shall have only 1 outgoing edge (transition), by specification and construction, as explained in Proposition 3 in Section 3.

The algorithm in Listing 3 demonstrates that, given correct trace and event information from SUT, `yr_sd_runtime_verif` always exactly matches the user specification. Thus never giving false warnings.

Table 5: SUT (YEROTH-ERP-3.0) Trace Output (Figure 5).

CONSOLE OUTPUT LINE	SQL EVENT	SUT PROGRAM POINT (TRACE)
21	"DELETE.department.YR_ASSET"	"src/admin/lister/yeroth-erp-admin-lister-window.cpp:1603"
22	"DELETE.merchandise.YR_ASSET"	"src/admin/lister/yeroth-erp-admin-lister-window.cpp:1626"
23	"SELECT.department"	"src/yeroth-erp-windows.cpp:967"

6 Evaluation

The main experimental results in this paper demonstrate the efficacy of our tool to find errors in the SUT (YEROTH-ERP-3.0), presented in Subsection 2.2.

Qualitative Results.

SUT (YEROTH-ERP-3.0) TRACING.

Table 5 illustrates SUT source code trace information as presented in **YR_DB_RUNTIME_VERIF** console output in Figure 5. We have translated from French to English the MariaDB SQL table names.

SQL EVENT CALL SEQUENCE.

A careful observation of the output in Figure 5 illustrates the following sequence:

- **line 23:** at state D , execution of the state diagram event "'SELECT.department' " (SUT button 'Delete' has been pressed at **line 21**)
:
select * from departements_produits WHERE nom_departement_produit = 'YR_ASSET';
- **line 28, line 29:** evaluation of the pre-condition Q_0 of state D stating that product department 'YR_ASSET' is not existent evaluates to 'TRUE' (triggering of event "'DELETE.department.YR_ASSET' " by pressing of SUT button 'Delete' at **line 21** has removed any asset department name 'YR_ASSET').
*[YR_CPP_MONITOR::CHECK_PRE_CONDITION_notIN:] precondition_IS_TRUE: True **
- **line 31, line 32:** checking post-condition $\overline{Q_1}$ in state E (there are still stocks in stock department 'YR_ASSET') evaluates to 'TRUE', thus state E is reached as an accepting state, because department name 'YR_ASSET' still exists in SUT SQL table "stocks", as illustrated in Figure 4 of the motivating example:
"execQuery: select * from stocks WHERE nom_departement_produit = 'YR_ASSET';"
*[YR_CPP_MONITOR::CHECK_post_condition_IN:] postcondition_IS_TRUE: True **

Runtime Performance.

YR_DB_RUNTIME_VERIF and **yr_sd_runtime_verif** don't incur a runtime supplemental overhead to the SUT, apart from emitting SQL events from SUT to **YR_DB_RUNTIME_VERIF** as they occur, since no hand-shaking mechanism is used between **YR_DB_RUNTIME_VERIF** and the SUT. The emission of an SQL event from SUT to **YR_DB_RUNTIME_VERIF** doesn't cost more than 2 statements execution time (getting a pointer to the DBUS server, and calling a method 'YR_slot_refresh_SELECT_DB_MYSQL' or other similar 3 methods (for INSERT, UPDATE, and, DELETE) on it).

7 Related Work

- **SUT source code instrumentation with runtime monitor specification.** "Clara" [12] enables to express software correctness properties using **AspectJ** and *dependency state machines*, both as instances of the *typestate* formalism, a formalism that is merely used for checking correctness of programs by a static compilation (analysis) technique called *typestate checking*. The Clara framework weaves (instruments), and annotates a program with runtime monitors using **AspectJ**, then tries to optimize the weaved program by static analysis. The "residual program", meaning the weaved statically optimized program is then executed and runtime monitored by developers to detect runtime errors. Runtime monitoring tools [13–16] work as similar as the Clara framework does.

YR_DB_RUNTIME_VERIF doesn't instrument the System Under Test (SUT) with any specification. It runs the runtime monitor concurrently from the analyzed SUT, but not with hand-shaking mechanism, thus not increasing runtime execution of the SUT. **YR_DB_RUNTIME_VERIF** specifies the runtime monitor as a state diagram mealy machine, a subset of *typestate*, specified as a C++ program, and extended with accepting states and state transition pre- and post-condition.

- **SUT binary code instrumentation with a runtime monitor.** With **tracerory** [6, 21], Jon Eyolfson and Patrick Lam use runtime program binary code instrumentation technique in **INTEL-pin** [22] to instrument running programs for purposes of detecting unread memory. I.e., **tracerory** doesn't generate itself a runtime monitor, it uses **INTEL-pin** [22] to generate a runtime monitor for its verification purposes. "Purify" [23] doesn't allow for SUT user correctness property specification. It has built-in memory access safety properties to check *offline* on program execution, after instrumentation of the SUT, its third-party, and vendor object-code libraries.

In contrast, with **YR_DB_RUNTIME_VERIF**, the user instruments the source code

of the analyzed C++ program at compile time with SQL events emitting code. **YR_DB_RUNTIME_VERIF** monitors program trace events at database level, and not at program counter level as **tracerory** does. **YR_DB_RUNTIME_VERIF** inputs a SUT correctness property specification as a state diagram (as a subset of LTL [2]).

- **Specification as set interface operations.** "Hob" [24, 25] is a program verification framework that enables to: characterize effects of program statement on data structures by means of all (\forall , \exists , etc.) algebra abstract set interface operations; and to check that these characteristics hold or not, using static analyses.

YR_DB_RUNTIME_VERIF is a program verification framework that enables to: characterize effects of program statements (via SQL [4] (Structure Query Language) on database table columns by means of set interface operations (\in , \notin); and to check that these characteristics hold or not, using dynamic runtime analysis.

- **Concurrent Event Stream Analysis.**

"DejaVu" [26] enables to check safety temporal property expressed in **first-order past linear-time temporal logic (FO-PLTL)** for events that carry data. **DejaVu** inputs a trace log (*offline*) and a FO-PLTL formula, and outputs a boolean value for each position in the inputted trace. "LOGSCOPE" [27] checks, *offline*, software systems correctness properties expressed using a rule-based specification language over state machines. It is not very precise what type of state machine is created and processed. "LOGSCOPE" translates specifications into C++ monitors (that could carry data). "EventRaceCommander" [28] repairs in web applications (*online*), event race errors, a kind of safety error.

States diagrams specifications are implemented as C++ program monitors using C++ library **yr_sd_runtime_verif**. **YR_DB_RUNTIME_VERIF** outputs a developer given (by means of a callback function, as

seen in 'line 15' in Listing 2) string message¹ in case an accepting state was entered, and a trace event of YEROTH-ERP-3.0 leading to it. **YR_DB_RUNTIME_VERIF**'s monitors need not store data, as DeJaVu monitors must. **YR_DB_RUNTIME_VERIF** events also carry data (database table and column name, records quantity modified by current SUT event). Runtime monitors could be checked against programs written in any programming language or framework, as long as they emit necessary SQL events to **YR_DB_RUNTIME_VERIF**.

¹'YR_DB_RUNTIME_VERIF_Monitor_notify_SUCCESS_VERIFICATION'
in this paper motivating example in Figure 5.

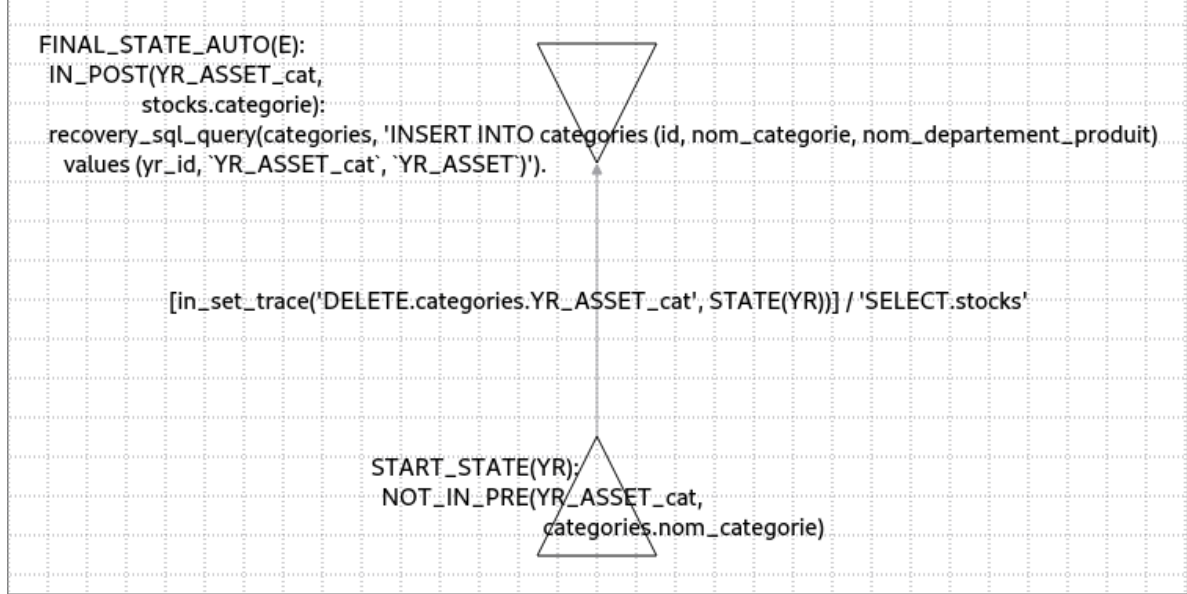
Fig. 11: A Mealy Machine State Diagram Specified Using `yr_sd_runtime_verif` Specification Language.

```

1. yr_sd_mealy_automaton_spec yr_missing_department
2. {
3.   START_STATE(d):NOT_IN_PRE(YR_ASSET,department.department_name)
4.   ->[in_sql_event_log('DELETE.department.YR_ASSET',STATE(d))]/'SELECT.department'->
5.     ERROR_STATE(e):IN_POST(YR_ASSET,stocks.department_name).
6. }

```

Fig. 12: 'YR_QVGE' model for the example specification in Figure 11.



8 Conclusion And Future Work

This paper has presented a lightweight C++ Qt-DBus [29] tool to check a program against a runtime monitor using set interface operations (\in , \notin) on program statement: **YR_DB_RUNTIME_VERIF**. **YR_DB_RUNTIME_VERIF** doesn't generate false warnings; **YR_DB_RUNTIME_VERIF** specifications are *not desirable (forbidden) specifications (fail traces)*. Since the concurrent communication between **YR_DB_RUNTIME_VERIF** and a program occurs over the RPC (Remote Procedure Call) instance **DBus**, a runtime monitor could be checked against programs written in any programming language or framework, as long as they emit the necessary SQL events to **YR_DB_RUNTIME_VERIF**.

Future work would be a tool-chain to validate `yr_sd_runtime_verif` models as represented in this paper.

Also, the author of this paper has developed a graphical drawing tool (YR_QVGE) for in Section 3 defined state diagrams. A model of YR_QVGE is shown in Figure 12. It is an extension of the FOSS (Free and Open Source Software) Qt Graphviz [30] drawing tool QVGE [31]. YR_QVGE generates, from a model, an input file for the compiler `yr_sd_runtime_verif_lang_comp`.

9 Acknowledgments

The author of this paper thanks Jan Peleska, and Thomas Ndie Djotio for helping him in continuing this research.

Listing 4: 'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY': YR_DB_RUNTIME_VERIF's overridden method for processing SUT event stream C++ pseudo-code.

```

1  bool DO_VERIFY_AND_or_CHECK_ltl_PROPERTY(
2      QString sql_table_NAME,
3      SQL_CONSTANT_IDENTIFIER cur_SQL_command)
4  {
5      switch (cur_SQL_command)
6      {
7          case SELECT:
8
9              if ("department" == sql_table_NAME))
10             {
11                 return YR_trigger_an_edge_event("'select.department'");
12             }
13             break;
14
15         default:
16             break;
17     }
18
19     return false;
20 }

```

A Processing of SUT Event Stream By An Analysis Client

B YR_SD_RUNTIME_VERIF SPECIFICATION LANGUAGE

Listing 4 illustrates the pseudo-code of YR_DB_RUNTIME_VERIF SUT event processing method 'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY'. An analysis client must first override method 'DO_VERIFY_AND_or_CHECK_ltl_PROPERTY' of class 'YR_DB_RUNTIME_VERIF_Monitor' so to implement a checking algorithm for each event received from SUT, as for instance the events illustrated in Figure 2 of the motivating example.

The analysis client then calls method 'YR_trigger_an_edge_event(QString an_edge_event)' of class 'YR_CPP_RUNTIME_MONITOR' of C++ library yr_sd_runtime_verif for each corresponding state diagram transition event.

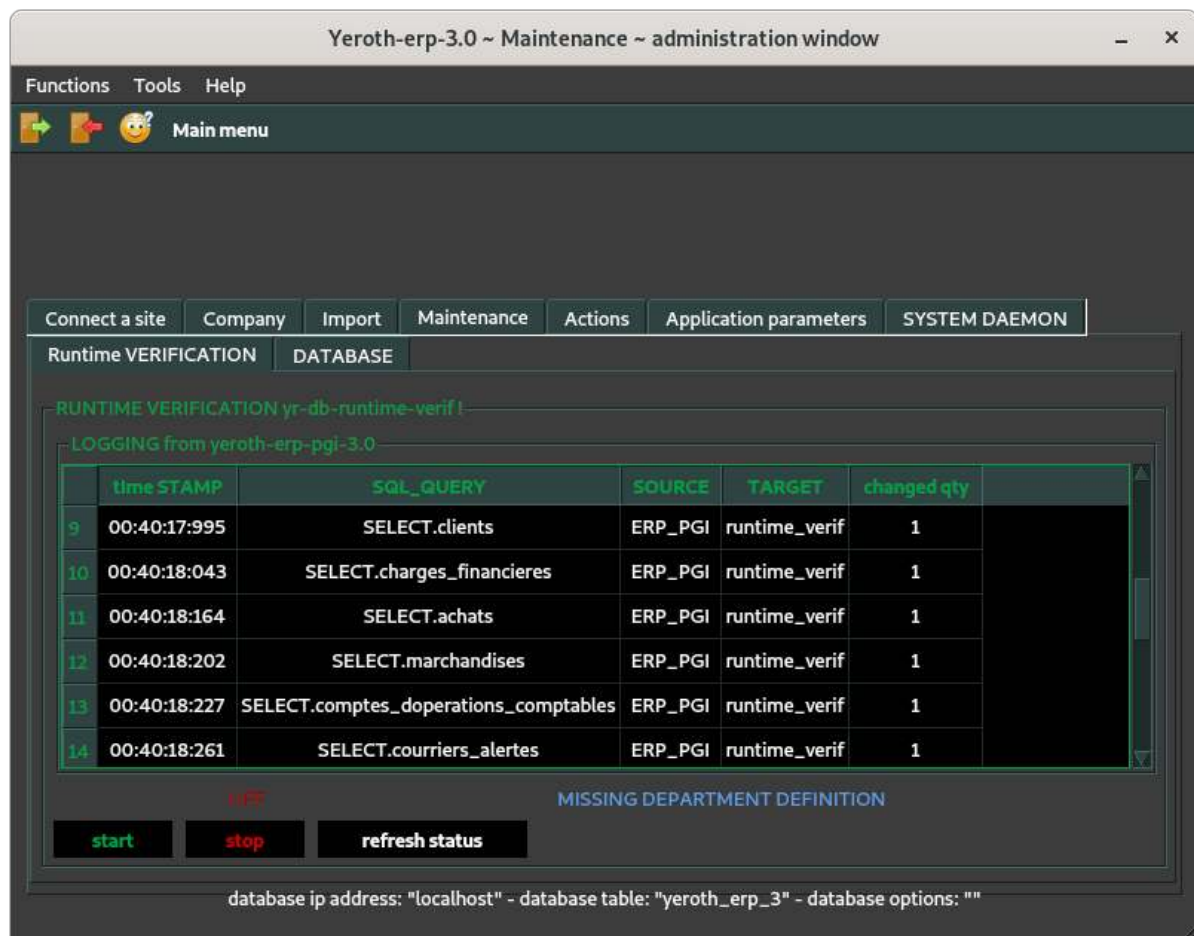
Fig. 13: Grammar in Backus–Naur Form (BNF) of `yr_sd_runtime_verif` Mealy Machine State Diagram Specification Language.

```

<specification> ::= yr_sd_mealy_automaton_spec '{' <mealy-automaton-spec> '}'
<mealy-automaton-spec> ::= <sut-state-spec>
                        | <sut-state-spec> '→' <sut-edge-state-spec>
<sut-edge-state-spec> ::= <sut-edge-mealy-automaton-spec> '→' <mealy-automaton-spec>
<sut-edge-mealy-automaton-spec> ::= <edge-mealy-automaton-guard-cond> <event-call>
<edge-mealy-automaton-guard-cond> ::= /* empty */ '/' | '[' <trace-specification> ']' '/'
<trace-specification> ::= <in-sql-event-log> | <not-in-sql-event-log> | <in-set-trace> | <not-in-set-trace>
<sut-state-spec> ::= <start-state-property-spec>
                  | <start-state-property-spec> ':' <algebra-set-specification>
                  | <state-property-spec> ':' <algebra-set-specification>
                  | <final-state-property-spec> ':' <algebra-set-specification>
                  | <final-state-auto-property-spec> ':' <algebra-set-specification> ':'
                  | <recovery-sql-query-spec>
<algebra-set-specification> ::= <in-algebra-set-spec> | <not-in-algebra-set-spec>
<in-algebra-set-spec> ::= <in-spec> '(' <prog-variable> ',' <db-table> ':' <db-column> ')'
<not-in-algebra-set-spec> ::= <not-in-spec> '(' <prog-variable> ',' <db-table> ':' <db-column> ')'
<in-sql-event-log> ::= in_sql_event_log '(' <event-call> ',' <state-property-specification> ')'
<not-in-sql-event-log> ::= not_in_sql_event_log '(' <event-call> ',' <state-property-specification> ')'
<in-set-trace> ::= in_set_trace '(' <event-call> ',' <state-property-specification> ')'
<not-in-set-trace> ::= not_in_set_trace '(' <event-call> ',' <state-property-specification> ')'
<in-spec> ::= IN_BEFORE | IN_AFTER
            | IN_PRE | IN_POST
<not-in-spec> ::= NOT_IN_BEFORE | NOT_IN_AFTER
              | NOT_IN_PRE | NOT_IN_POST
<start-state-property-spec> ::= START_STATE '(' AlphaNum ')'
<state-property-spec> ::= STATE '(' AlphaNum ')'
<final-state-property-spec> ::= END_STATE '(' AlphaNum ')'
                           | FINAL_STATE '(' AlphaNum ')'
                           | ERROR_STATE '(' AlphaNum ')'
<final-state-auto-property-spec> ::= END_STATE_AUTO '(' AlphaNum ')'
                                | FINAL_STATE_AUTO '(' AlphaNum ')'
                                | ERROR_STATE_AUTO '(' AlphaNum ')'
<recovery-sql-query-spec> ::= recovery_sql_query '(' <db-table> ',' <sql-recovery-query> ')'
<sql-recovery-query> ::= String
<event-call> ::= String
<prog-variable> ::= AlphaNum
<db-table> ::= AlphaNum
<db-column> ::= AlphaNum

```

Fig. 14: YEROTH-ERP-3.0 Maintenance Verification Interface.



C YEROTH-ERP-3.0 MAINTENANCE VERIFICATION INTERFACE

References

- [1] Wikipedia.org: SQL - Wikipedia. <https://en.wikipedia.org/wiki/SQL>. Accessed last time on February 08, 2023 at 12:00 (2023)
- [2] Clarke, E.M., Grumberg, O., Kroening, D., Peled, D.A., Veith, H.: Model Checking, 2nd Edition. (2018). <https://mitpress.mit.edu/books/model-checking-second-edition>
- [3] Wikipedia.org: Mealy machine. https://en.wikipedia.org/wiki/Mealy_machine. Accessed last time on Dec 15, 2022 at 12:00 (2022)
- [4] MariaDB.org: MariaDB Foundation - MariaDB.org. <https://www.mariadb.org>. Accessed last time on June 24, 2022 at 12:20 (2022)
- [5] Noundou, X.N.: YEROTH-ERP-PGI-3.0 Doctoral Compendium. https://archive.org/download/yeroth-erp-pgi-compendium_202206/JH_NISSI_ERP_PGI_COMPENDIUM.pdf. Accessed last time on January 21, 2023 at 23:24 (2022)
- [6] Eyolfson, J., Lam, P.: Detecting unread memory using dynamic binary translation. In: Qadeer, S., Tasiran, S. (eds.) Runtime Verification, pp. 49–63. Springer, Berlin, Heidelberg (2013)
- [7] Bergenthal, M., Krafczyk, N., Peleska, J., Sachtleben, R.: libfsmtest an open source library for fsm-based testing. In: Clark, D., Menendez, H., Cavalli, A.R. (eds.) Testing Software and Systems, pp. 3–19. Springer, Cham (2022)
- [8] doc.qt.io/qt-5: Qt 5.15. <https://doc.qt.io/qt-5>. Accessed last time on Dec 22, 2022 at 12:40 (2022)
- [9] <https://freedesktop.org/wiki/Software/cppunit:cppunit>. <https://doc.qt.io/qt-5/qtdbus-index.html>. Accessed last time on January 01, 2023 at 12:00 (2022)
- [10] Alpern, B., Ngo, T., Choi, J.-D., Sridharan, M.: DeJaVu: deterministic Java replay debugger for Jalapeño Java virtual machine. In: Addendum to the Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2000). <https://doi.org/10.1145/367845.368073>
- [11] froglogic.com: Home • froglogic. <https://www.froglogic.com/home>. Accessed last time on Dec 18, 2022 at 20:00 (2022)
- [12] Bodden, E., Hendren, L.: The clara framework for hybrid typestate analysis. International Journal on Software Tools for Technology Transfer (STTT) **14**, 307–326 (2012). 10.1007/s10009-010-0183-5
- [13] Butkevich, S., Renedo, M., Baumgartner, G., Young, M.: Compiler and tool support for debugging object protocols. In: SIGSOFT ’00/FSE-8 (2000)
- [14] Allan, C., Avgustinov, P., Christensen, A.S., Dufour, B., Goard, C., Hendren, L.J., Kuzins, S., Lhoták, J., Lhoták, O., Moor, O., Sereni, D., Sittampalam, G., Tibble, J., Verbrugge, C.: abc the aspectbench compiler for aspectj a workbench for aspect-oriented programming language and compilers research. In: Johnson, R.E., Gabriel, R.P. (eds.) Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16–20, 2005, San Diego, CA, USA, pp. 88–89. ACM, ??? (2005). <https://doi.org/10.1145/1094855.1094877>. <https://doi.org/10.1145/1094855.1094877>
- [15] Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Diploma thesis, RWTH Aachen University (November 2005). <https://www.bodden.de/pubs/bodden05jlo.pdf>
- [16] Chen, F., Rosu, G.: Mop: an efficient and generic runtime verification framework. In: Gabriel, R.P., Bacon, D.F., Lopes, C.V., Jr., G.L.S. (eds.) Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages and Applications, pp. 569–588. ACM, ??? (2007). <https://doi.org/10.1145/1297027.1297069>

- [17] Noundou, X.N.: Yr_db_runtime_verif: a framework for verifying sql correctness properties of gui software at runtime (2023). https://archive.org/download/yr_ictss_2023/yr_ictss_2023.pdf
- [18] Peleska, J., Huang, W.-l.: Test Automation; Foundations and Applications of Model-based Testing. <https://www.informatik.uni-bremen.de/agbs/jp/papers/test-automation-huang-peleska.pdf>. Accessed last time on May 06, 2023 at 12:00 (2021)
- [19] Harel, D.: Statecharts: a visual formalism for complex systems. *Science of Computer Programming* **8**(3) (1987)
- [20] Booch, G., Rumbaugh, J., Jacobson, I.: Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series). (2005)
- [21] Eyolfson, J.: Tracerory; Dynamic Trace-matches and Unread Memory Detection for C/C++. (2012). MASTER OF APPLIED SCIENCES (MAsc). <https://hdl.handle.net/10012/6206>
- [22] Luk, C.K., Cohn, R.S., Muth, R., Patil, H., Klauser, A., Lowney, P.G., Wallace, S., Reddi, V.J., Hazelwood, K.M.: Pin: building customized program analysis tools with dynamic instrumentation. In: *PLDI '05* (2005)
- [23] Hastings, R.O., Joyce, B.A.: Fast detection of memory leaks and access errors. (1991)
- [24] Kuncak, V., Lam, P., Zee, K., Rinard, M.: Modular pluggable analyses for data structure consistency. *Transactions on Software Engineering* **32**(12), 988–1005 (2006)
- [25] Lam, P.: The Hob System for Verifying Software Design Properties. (2007)
- [26] Havelund, K., Peled, D., Ulus, D.: Dejavu: A monitoring tool for first-order temporal logic, pp. 12–13 (2018). <https://doi.org/10.1109/MT-CPS.2018.00013>
- [27] Havelund, K.: Specification-based monitoring in c++. In: Margaria, T., Steffen, B. (eds.) *Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, pp. 65–87. Springer, Cham (2022)
- [28] Adamsen, C.Q., Møller, A., Karim, R., Sridharan, M., Tip, F., Sen, K.: Repairing event race errors by controlling nondeterminism. In: *Proceedings of the 39th International Conference on Software Engineering, ICSE (2017)*. <https://doi.org/10.1109/ICSE.2017.34> . files/ICSE17Repairing.pdf
- [29] doc.qt.io/qt-5/qtdbus-index.html: Qt D-Bus. <https://doc.qt.io/qt-5/qtdbus-index.html>. Accessed last time on Dec 22, 2022 at 12:40 (2022)
- [30] graphviz.org: DOT Language | Graphviz. <https://graphviz.org/doc/info/lang.html>. Accessed last time on JUNE 8, 2022 at 12:30 (2022)
- [31] showroom.qt.io: QVGE; Qt Visual Graph Editor | Showroom. <https://showroom.qt.io/qvge-qt-visual-graph-editor>. Accessed last time on Jun 27, 2022 at 12:40 (2022)